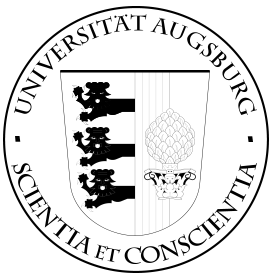
Dynamic Coarse Grained Reconfigurable Architectures

DISSERTATION

FOR THE DEGREE OF DOCTOR OF ENGINEERING (DR.-ING.)

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE OF THE UNIVERSITY OF AUGSBURG



PRESENTED BY

BASHER SHEHAN

IN 2010

1. Reviewer: Prof. Dr. Theo Ungerer

2. Reviewer: Prof. Dr.-Ing. Rudi Knorr

Day of the defense: 23.11.2010

Signature from head of PhD committee:

Abstract

Coarse grained reconfigurable processors have gained more popularity in the last years, as they introduce a new way for a dynamic and programmable execution similar to FPGA and tend to achieve the performance of application specific hardware. The reconfigurability on instruction level grants these architectures a big dynamicity and ability to embrace the diversity of the applications. Nevertheless, managing the hardware resources in the software prevents from undertaking many dynamical reactions needed by the reconfiguration task at runtime to be adaptive to the dynamic program execution. However, an adaptive architecture can face the diversity of applications dynamically in the hardware without any software manipulation. On the other hand, the need for more flexibility to manage the underlying hardware structures increases the demands on the configuration hardware unit.

This work focuses on the design and optimization of reconfigurable coarse grained processors. In addition, it concerns with the implementation of the configuration task in the hardware. The **G**rid **A**lu **P**rocessor (GAP) is presented as baseline architecture for the design and optimization issues. We combine the characteristics of superscalar processors and coarse grained reconfigurable architectures to achieve a dynamicity and performance beyond that of out-of-order superscalar processors. Hence, the GAP comprises an in-order superscalar frontend and reconfigurable backend. A special config-

uration unit—fully integrated into the processor frontend instead of the issue stage—dynamically maps a conventional instruction stream at runtime to an array of reconfigurable functional units (FUs) inside the grid.

A very important feature of our researched design is that it does not require a new ISA and special software or controlling processor to prepare and map the configurations to the hardware. It permits herewith the use of the well-known GCC compiler for superscalar architectures without any modifications of the generated binary files. To that, the in-order and simultaneous reconfiguration of dependent and independent instructions at runtime keeps the processor front-end simple and avoids the most large, unscalable hardware structures needed by out-of-order processors, for example: large issue windows and the needed hardware to control it, renaming structures, and reorder buffer.

Acknowledgements

I would like to acknowledge the contribution of several individuals to the completion of this dissertation. First, I thank my advisor Prof. Dr. Theo Ungerer for his excellent mentoring to do a high-quality research in computer architecture and for instilling in me the confidence to become an independent thinker and do research on my own. This dissertation would certainly not be possible without his dedication and hard work. I would also like to thank Prof. Dr. Sascha Uhrig for his technical contributions to this dissertation. He was the administrator of the GAP project and his ideas were the trigger to apply to the project funding of *"Deutsche Forschungsgemeinschaft*" (DFG).

I thank also Prof. Dr.-Ing. Rudi Knorr for his dedication to my writing and Prof. Dr. Elisabeth André and Prof. Dr. Bernhard Bauer for accepting my request for being examiner. My thank goes also to all my colleagues at the Department of Systems and Networking at the University of Augsburg for their support, discussions, and comments on my work especially Ralf Jahr, who join me the work on this project. I am also graceful to the DFG for providing the funding for this project.

Finally, I thank my family for their constant support throughout my academic career, and for pushing me to pursue my ambitions.

Ehrenwörtliche Erklärung zu meiner Dissertation

mit dem Titel: .....

Sehr geehrte Damen und Herren,

hiermit erkläre ich, dass ich die beigefügte Dissertation selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel genutzt habe. Alle wörtlich oder inhaltlich übernommenen Stellen habe ich als solche gekennzeichnet.

Ich versichere außerdem, dass ich die beigefügte Dissertation nur in diesem und keinem anderen Promotionsverfahren eingereicht habe und, dass diesem Promotionsverfahren keine endgültig gescheiterten Promotionsverfahren vorausgegangen sind.

....

Ort, Datum Unterschrift

dedicated to my family

Contents

Chapter 1

Introduction

Currently, few architectural approaches are proposing new paths to raise the performance of conventional sequential instruction streams. Over the last thirty years, microprocessor industry has relied on instruction level parallelism and aggressive clock scaling to keep the steady gain of performance defined by Moore's Law. Hardware complexity as well as power consumption has been increased dramatically with more pipelining, which opens the door for multicore architectures. In the time of the billions transistor era, hardware designers succeeded to execute several threads simultaneously on multicore systems moving toward high thread level parallelism (TLP). However, these architectures can not offer an acceleration for single threaded execution. Moreover, the multicore processors have inherited many hurdles from single core like memory wall [1] and sequential program execution and also incurred other difficulties of synchronization and communication.

Nevertheless, asymmetric multiprocessor (AMP) design could offer an underlying fabric that optimally meets the demands of diverse applications. However, these architectures need a lot of complex compiler analysis and task mapping mechanisms. Moreover, the application performance depends again on the efficiency of the cores. Application specific integrated processors (ASIPs) allow herewith the finding of an op-

timal hardware solution for a special kind of applications. Unfortunately, ASIPs usually target a narrow class of applications and exhibit a very poor dynamicity regarding the divers of the applications. Therefore, the reconfiguration of processors on instruction level came up to offer a single design that effectively executes different applications with different demands [2]. These architectures can reconfigure a grid of processing elements on the instruction level offering a superior dynamicity and tend to achieve the performance of ASIPs.

In the last decade, many coarse grained reconfigurable processors have been introduced to copes with the challenge of the application diversity and offer a dynamic execution for different code structures of the applications. The previous attempts to achieve reconfigurability on the instruction level are mostly based on compiler analysis and profiling of the data flow graphs of the programs [3] [4]. A main processor with extended instruction set architecture (ISA) controls the mapping of specified tiles of program code to an accelerator [5] [6] [7] [8]. The underlying hardware structures on the grid determine the complexity of the reconfiguration task. However, implementing the configuration task in the software increases the demands on the grid to be able to dynamically redirect the results between the processing elements at runtime and to take the interconnection occupancy into account. Indeed, the cooperation between hardware and software and the need for a controlling processor has decreased the expectation of much better performance. Accelerators suffer also a delay resulting from the routing time needed between the processing elements on the grid. To that, the memory organization turns out to be the most crucial issue for coarse grain architectures, as it is necessary to provide the operations that are executing inside the grid with necessary data from the memory. Unfortunately, the memory subsystem in coarse grain architectures mostly increases the complexity and hampers the performance of the design.

1.1 The Contribution of this Research

This work is focusing on the dynamic coarse grained reconfigurable architectures. Our studies are presented by implementing the design and optimization issues to the **G**rid **A**lu **P**rocessor (GAP). The GAP is a runtime reconfigurable processor designed for the acceleration of a conventional sequential instruction stream without the need of recompilation. In contrast to previous attempts of extracting the configurations in the software, we integrate the reconfiguration task into the hardware as pipeline stage in the GAP processor. The hardware reconfiguration of an array of functional units allows the deployment of well-known GCC-compiler with a common RISC instruction set without any modifications. The array can save the configurations of loops and functions and execute them without the need for further fetching and reconfiguration of the already mapped code structures. This results in a high instruction level parallelism beyond that of out-of-order processors. A further improvement of the performance is also achieved by designing a grid that asynchronously executes the configured instructions, whereas other stages of the processor execute synchronously. To that, the configuration unit is able to map dependent and independent instructions simultaneously and at runtime into the grid in order to speed up the execution of critical paths, especially in sequential programs.

Another dimension in the coarse grained architecture is introduced by implementing several configuration layers attached to each reconfigurable functional unit. Configuration layers aim at storing the computational extensive part of the program near the execution units, which reduces the I-cache accesses and misses, leads to better execution times, and relieves the front-end of the processor. Several other design optimizations are exhaustively researched and presented to offer an effective design with less hardware costs and to enhance the performance. The number of functional units in the grid, a better utilization, and the interconnections are the main targets of optimizations. Moreover, the implemented optimizations take into account the asynchronous constraints with the

simple forwarding of results between functional units inside the grid.

We adapt our researched architecture copes with the challenge of accelerating sequential applications by:

- Mapping dependent and independent instructions simultaneously and at run time to the array of FUs and execute them asynchronously, which speeds up the execution of the critical paths inside the sequential applications.
- Asynchronously executing the loops and functions inside the array, which result additionally in a high instruction level parallelism (ILP).
- Parallelizing the memory accesses especially inside the loops and functions, where the memory access instructions are already mapped to the array.

The ILP in GAP processor is exploited by many factors, including:

- The array can execute an already mapped loop body that contains mostly more than four instructions.
- Layers of configurations also increase the number of instructions ready to execute in the array (nested loops and functions).
- Many load/store units expand the parallelism of the basic blocks vertically and parallelize the memory accesses especially with data intensive applications.

In this work, we discuss all of these characteristics in conjunction with the hardware design implementation. To this, we address the memory organization of coarse grained designs. A special cache access scheme with memory disambiguation mechanism is introduced to preserve the exploited parallelism inside the grid. The presented scheme keeps the consistency of the memory without any compiler optimization or extra dependency prediction mechanism. Additionally, the access scheme is enhanced to predict the accesses to the memory.

1.2 The Organization of the Work

In **Chapter 2**, we give an overview over the state of the art technologies. The effectiveness and the limitations of each technology are discussed to draw some conclusions and to motivate to our contribution. **Chapter 3** discusses the above listed characteristics and the detailed design issues of the GAP processor. The optimizations done in **Chapter 4** of this work concern the tuning of the hardware inside the grid and extending the configuration task to be able to reduce the overall hardware costs and enhancing the performance. Many design solutions targeting the design of the processing elements, the hardware specialization, the interconnections, and the mapping strategy are presented. **Chapter 5** discusses the memory subsystem design and disambiguation issues of the GAP processor with special memory access scheme. A simple and completely in hardware implemented memory access mechanism is presented to enable a simple memory disambiguation and parallelized memory accesses. Finally we draw some conclusions of this work in **Chapter 6** and give an outlook for future works.

Chapter 2

State of the Art Technologies

In this chapter, we discuss the background of some technologies related to our research. We focus on the benefits and the bottleneck of each technology to show—in the next chapter 3—how we combine the architectural effectiveness of these technologies to build our design.

2.1 Superscalar Processors

Superscalar processing is an advanced microarchitectural design amid of several innovations in processor architecture to exploit a high instruction level parallelism (ILP) [9] [10]. By exploiting the ILP, superscalar processors are capable of executing more than one instruction in a clock cycle. These architectures are widely spread in the early 90's of the last century and designed and produced by all the microprocessor vendors for high-end products (like the MIPS R10000, the AMD 29000-series, PowerPC 970, and Intel Pentium-Pro etc.) [11] [12].

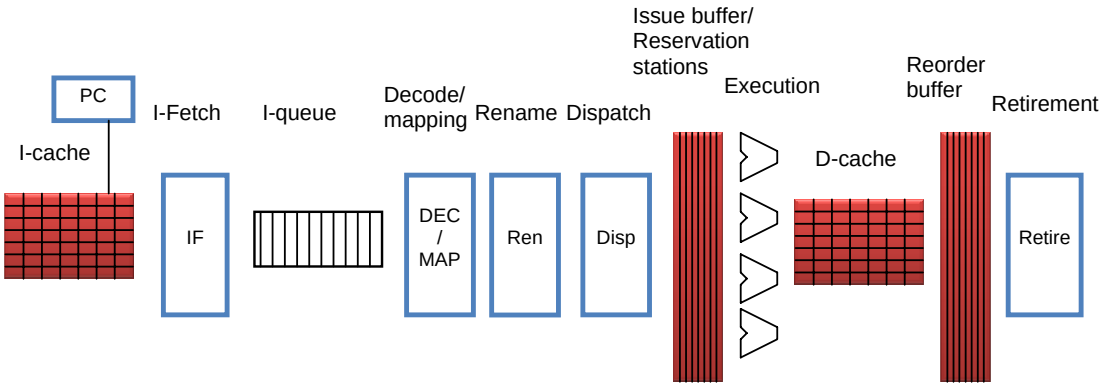


Figure 2.1: Superscalar Pipeline

2.1.1 Superscalar Architectural Design

Figure 2.1 shows a typical superscalar processor with the functionality of each pipeline stage. The major parts of the microarchitecture are: instruction fetch and branch prediction, decode and register dependence analysis, issue and execution, memory operation analysis and execution, and instruction reorder and retirement. A typical superscalar processor fetches and decodes several instructions at a time of the incoming instruction stream. As part of the instruction fetching process, the outcomes of conditional branch instructions are usually predicted in advance to ensure an uninterrupted stream of instructions. The incoming instruction stream is then analyzed for data dependences, and instructions are distributed to functional units, often according to instruction type. Next, instructions are initiated for execution in parallel, based primarily on the availability of operand data, rather than their original program sequence. For this purpose, an extra circuitry beside the issue buffer wakes up the instruction with data dependency when the execution of previous instructions –on which they depend—has been finished (scoreboarding). This important feature, present in many superscalar implementations, is referred to as dynamic instruction scheduling. Upon completion, instruction results are re-sequenced so that they can be used to update the process state in the correct (orig-

inal) program order in the event that an interrupt condition occurs. Because individual instructions are the entities being executed in parallel, superscalar processors exploit what is referred to as instruction level parallelism (ILP).

The general problem solved by superscalar processors is to convert an ostensibly sequential program into a more parallel one. The pipeline of a superscalar processor is capable of fetching and executing multiple instructions on each clock cycle. For these kind of functionalities, a specific implementation techniques are used in the important phases of superscalar processing. These major phases include:

- Instruction-fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions.
- Methods for determining true dependences involving register values, and mechanisms for communicating these values to where they are needed during execution.
- Methods for selecting and initiating, or out-of-order issuing, multiple instructions in parallel.
- Methods for communicating data values through memory via load and store instructions, and memory interfaces that allow for the dynamic and often unpredictable performance behavior of memory hierarchies. These interfaces must be well-matched with the instruction execution strategies.
- Methods for committing the process state in correct order; these mechanisms maintain an outward appearance of sequential execution.

The out-of-order issuing is mainly achieved by saving the instructions temporarily in instruction-issue buffer and extracting/issuing the instructions without data dependency to the functional units. The instruction-issue buffer implements, in essence, a limited data flow capability, in holding instructions while their operands are being generated,

and allowing ready instructions to issue out-of-order. The buffer may issue multiple instructions in a clock cycle to a number of functional units which operate concurrently. The operation of the instruction issue buffer can be split into two phases: wakeup and selection. The wakeup logic matches results generated by the functional units to the operands in the issue buffer, and the selection logic determines which of the ready instructions should be issued to free functional units. These architectures may issue dependent instructions in consecutive clock cycles by waking instructions in the same cycle as their final operand is being produced. A network of result buses and bypass logic is used to obtain the correct operand values on the subsequent clock cycle, which is commonly termed as data forwarding.

Reservation stations are usually used in conjunction with Tomasulo's algorithm [13]. This algorithm differs from scoreboarding by issuing the instructions in their sequence and using register renaming to eliminate RAW, WAR, and WAW hazards. The register renaming allows the continual issuing of instructions in the presence of data dependencies to achieve a high instruction-issue bandwidth.

2.1.2 Limitations of Superscalarity

Parallel instruction processing requires: the determination of the dependence relationships between instructions, adequate hardware resources to execute multiple operations in parallel, strategies to determine when an operation is ready for execution, and techniques to pass values from one operation to another. When the effects of instructions are committed, and the visible state of the machine updated, the appearance of sequential execution must be maintained. More precisely, in hardware terms, this means a superscalar processor implements complex circuitry structures to enable the elaboration on a sequential stream of instructions. The hardware costs as well as the hardware complexity escalates with the increasing of the issue bandwidth with the number of functional units. These complexities arise from data dependence, control hazard,

and resource conflict checking circuitry logic [14] [15]. On the other hand, the more pipelining has enabled to scale the clock cycle to the underlying development transistor technology [16] [17]. However, the more pipelining has tremendously increased the hardware cost/complexity and the power dissipation, which implicates performance limitations on superscalar computing.

2.2 Globally Asynchronous Locally Synchronous Architectures

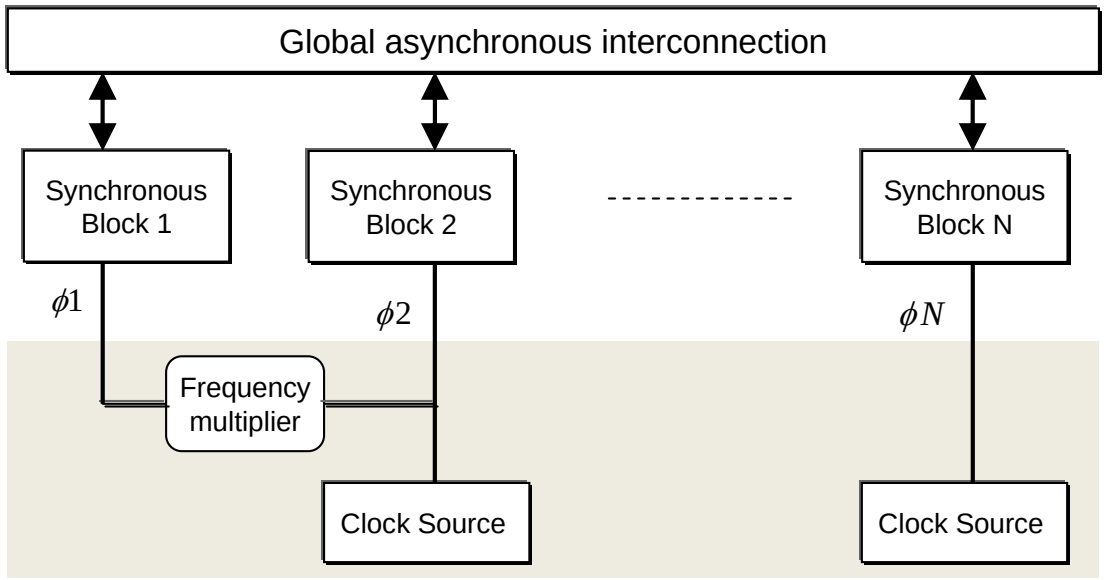


Figure 2.2: Block diagram of globally asynchronous locally synchronous system

The term of globally asynchronous locally synchronous (GALS) is usually used when the design comprises multiple clock domains running at different frequencies with a global asynchronous synchronization bus [18] [19] [20] [21], as shown in Figure 2.2. The GALS designs have gained their popularity with the steady improvement in semi-

conductor manufacturing technology and the increase of the number of devices that can fit on a single die. This has made it more difficult to design a global-clock network that can control all the blocks in the design, where such a network significantly increases the overall power consumption.

High performance microprocessors face similar pressures. As transistor counts and clock frequencies increase, distributing a low-skew global clock becomes increasingly more difficult. Different studies have focused on GALS-based microprocessors and concluded that they could gain power advantages by allowing fine tuning of the supply voltages and clock speeds for different functional blocks and by eliminating the need for a global, low-skew clock [22] [23] [24]. In these studies a superscalar processor is designed with multiple clock domains and using synchronizer/wrapper for the synchronization between the different blocks.

It is Obvious that simple and complex operations require different execution times in the arithmetic logical units (ALUs). Unfortunately, because of the synchronously clocked pipelines all operations in the execution stage consume the same time of execution, which is one clock cycle. The globally asynchronous locally synchronous architectures (GALS) remove the delay between different stages in the processor and retire the result as soon as the execution of the operation has been finished. Following a similar technique, we applied asynchronous execution to our design in the array of functional units, where all other stages are synchronous. This can be again seen as globally synchronous locally asynchronous (GSLA) [25], where GSLA and GALS designs are event driven synchronization methods.

Advantages of GALS design:

- A GALS approach can facilitate fast block reuse by providing wrapper circuits to handle interblock communication across clock domain boundaries. Moreover, it reduces the electromagnetic interferences.

- The capability to independently configure each domain to execute at frequency/voltage settings at or below the maximum values. This allows domains that are not executing operations critical to performance to be configured at a lower frequency. Consequently, a GALS microarchitecture has the advantage that power can be saved [26] [23] [24].

- Improving the average-case performance.

Drawbacks:

- It can produce an overhead in the communications between different domain blocks.

- The synchronization between the blocks requires additional circuitry that must be designed carefully to avoid metastability state. Additionally, the implementation of this circuit incurs a hardware overhead.

2.3 Reconfigurable Architectures

Reconfigurable computing as a concept has been in existence for about sixty years ago [27]. The reconfigurability in today's microarchitectural designs has grown to a wide term. A part or all the components of the design can be reconfigured statically or dynamically at runtime to adapt the variations during the execution. Even general-purpose processors use some of the same basic ideas, such as reusing computational components for independent computations, and using multiplexers to control the routing between these components. However, the term reconfigurable computing, as it is used in current research refers to systems incorporating some form of hardware programmability to realize a special functionality.

Reconfigurable designs can be mainly classified into fine grain and coarse grain ar-

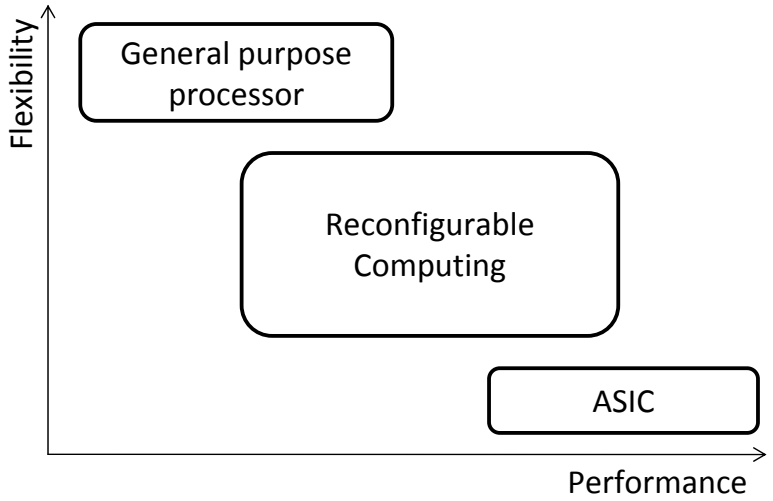


Figure 2.3: Reconfigurable computing vs. general purpose or ASIC design

chitectures. Fine grain devices can be reconfigured on the bit level, whereas complete processing elements can be reconfigured in coarse grain devices. As shown in Figure 2.3 reconfigurable platforms are heading to mainstream domain, bridging the gap between ASICs and general purpose microprocessors [2].

2.3.1 Fine Grain Architectures

Fine grain reconfigurable architectures are usually mentioned to as field programmable gate arrays (FPGAs), that are widely spread in today's designs. These devices are fully electrically programmable, meaning that the physical design costs are amortized over multiple application circuit implementations. FPGAs contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements are connected using a set of routing resources that are also programmable. In this way, custom digital circuits can be mapped to the reconfigurable hardware by computing the logic functions of the circuit within the logic blocks, and using the configurable routing to connect the blocks together to form the necessary circuit.

Additional routing, synthesis, scheduling, and placement tools are used in conjunction to these devices to control the correct mapping and execution.

In contrast to ASICs—where the circuit cannot be altered after fabrication—FPGAs can realize several required circuitries while maintaining a higher level of flexibility. However, fine grain architectures are much less efficient because of the huge routing area overhead and the poor routability [28]. Nevertheless, this area of reconfigurable computing has shown encouraging results in a number of application areas including cryptography, signal processing, and searching.

In order to achieve these performance benefits, yet support a wide range of applications, reconfigurable systems are usually formed with a combination of reconfigurable logic and a general purpose microprocessor. The processor performs the operations that cannot be done efficiently in the reconfigurable logic, such as data-dependent control and possibly memory accesses, while the computational cores are mapped to the reconfigurable hardware [29] [30] [31] [32] [33]. For this purpose, a special compilation environment must be developed to transform the code into synthesized FPGA hardware circuit. The design process involves first partitioning a program into sections to be implemented on hardware, and those which are to be implemented in software on the host processor.

2.3.2 Coarse Grain Architectures

Coarse grain architectures [34] [35] [2] enable a reconfigurability on the word level instead of the bit level reconfiguration in fine grain architectures. The data path in coarse grain design is equipped with specialized processing elements (PEs) and fast interconnections as shown in Figure 2.4. Since computational data paths have regular structure, full custom designs of reconfigurable PE can be drastically more area-efficient, than by assembling the FPGA way from single-bit matrix of configurable logic blocks

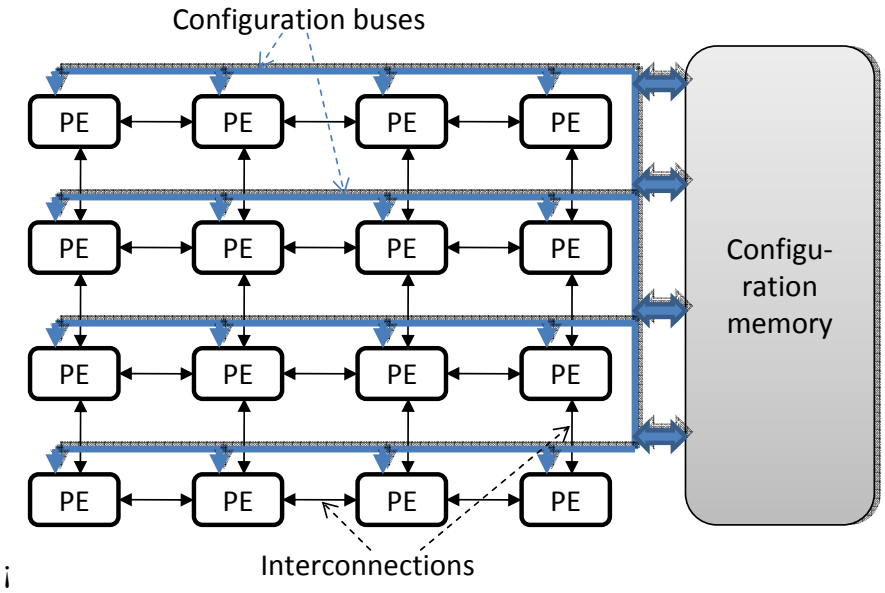


Figure 2.4: Reconfigurable computing vs. general purpose or ASIC design

(CLBs). With the help of synthesis software-tools/compiler, these PE can be reconfigured to form a data-flow-similar accelerator. Coarse-grained architectures provide operator level configurable functional blocks (CFBs), word level data paths, and powerful and very area-efficient data path routing switches. The motivating implementation of such architectures is to deploy a single hardware structure that can optimally execute different applications with diverse code snippets as in ASICs. An optimal placement of the configurations in the grid of the processing element leads to a very fast execution of the workloads. In addition, a main control processor is usually used in conjunction with the reconfigurable grid to execute the non-loop sequential code, control the mapping of configurations to the grid, and supervise the execution activities.

Coarse grain architectures trade off programming flexibility for more efficient reconfigurable hardware. Reducing the programming flexibility has a direct impact on the configuration size and, subsequently, on configuration latency and on reconfiguration overhead and as well as drastic complexity reduction of the P&R (placement and routing) problem. Thus, loading a decoding object that occupies a tenth of a VIRTEX

XC2V6000 FPGA [36] involves loading a configuration of 260 KB (using partial reconfiguration) with a reconfiguration latency of 4 ms when clocking the configuration bus at the maximum speed. The configuration size of the same task for a coarse grain array can be between 10 and 50 times smaller depending on the programming granularity [37]. Of course for fine-grain the decoding object can be optimized at bit level, whereas for coarse-grain it must be implemented using operations of a fixed bit width and normally with less interconnection possibilities. However, if the coarse-grain architecture fits appropriately the decoding computations, it will provide good performance and fast reconfigurations.

The number of reconfigurable devices has rapidly grown in the last decade to diverse architectures with different characteristics, where recently some of them are applying reconfigurability to multicore designs [38]. *R.Hartenstein* [2] has divided coarse grain architectures into three categories: mesh-based, linear-array-based and crossbar-based architectures. However, we distinguish mainly between FPGA-based and ASIC-based target-designs, since we mainly target ASIC-based architectures. FPGA-based architectures introduce multi-granular solutions, where more coarse granularity can be achieved by bundling of resources, such as e. g. 4 ALUs of 4 bits each to obtain a 16 bit ALU. FPGA accelerators are usually tightly coupled to a host processor, where specific parts of the program are transferred to a data path to be executed on customized reconfigurable data path units (like DP-FPGA [39],MOLEN [7],WARP [40]). This work is more related to ASIC-based target-designs, and hence, we focus here more on ASIC-based architectures.

Mesh-based architectures arrange their PEs in a rectangular 2-D array with horizontal and vertical connections which supports rich communication resources for efficient parallelism. The hardware structure of the PE differs from simple ALU to complete processor with routing elements or multiplexers and control registers. The connectivity to nearest neighbor (NN) links between adjacent PEs (NN or 4NN: links to 4 sides east. west. north, south, or, 8NN: NN-links to 8 sides east, northeast, north, north-west, west,

south-west, south, south-east, like in CHESS array [41]). Typically, longer lines are added with different lengths for connections over distances larger than one to increase the connectivity of the resources (like in MorphoSys [42]).

Architectures based on linear array or several linear arrays are typically implemented with an NN connect. they aim at mapping executable pipeline instructions onto it. Additional routing resources are usually used—like longer lines spanning the whole or a part of the array—if the pipes have forks, which otherwise would require a 2-D realization. Two Reconfigurable architectures have linear array structure (RaPiD, PipeRench). RaPiD provides different computing resources, like ALUs, RAMs, multipliers, and registers, but irregularly distributed. While RaPiD uses mostly static reconfiguration, PipeRench relies on dynamic reconfiguration, allowing the reconfiguration of a PE in each execution cycle. Besides the mostly unidirectional NN connects, it provides also a global bus.

Crossbar-based architectures support the communication between the PE with crossbar switches. A full crossbar switch features a most powerful communication network that is easily to route. The architectures of this category discussed here use only reduced crossbars. PADDI-1 [43] is developed for fast prototyping of DSP datapaths and features eight PEs, all connected by a multilayer crossbar. PADDI-2 [44] has 48 PEs, but saves area by restricted crossbar with a hierarchical interconnect structure for linear arrays of PEs forming clusters. This fabric sophistication has again an impact on routing.

Several architectures are briefly outlined as shown in Table 2.1. In the next we give a short description of their architectures.

MorphoSys (Morphoing System [5] [42] [45]) comprises a TinyRISC (MIPS-like) processor with extended instruction set, a 8 by 8 connected grid, a frame buffer for intermediate data, context memory, and DMA controller. TinyRISC with extra DMA instructions initiate data transfers between the main memory and the frame buffer (inter-

nal data memory for blocks of intermediate results). The reconfigurable components on the grid are connected by the means of two layers of interconnection networks. The first layer connects the components with a 2-D mesh, whereas the second layer hardwires the quadrant level to provide a complete row and column connectivity within the quadrant. For the data transportation a dedicated 128-bit bus is linked to the column elements on the array. A context bus is also deployed to distribute the required instructions to the processing elements. The grid is divided into four quadrants of 4 by 4 16 bit PEs each, featuring ALU, multiplier, shifter, register file, and a 32 bit context register for storing the configuration word.

TRIPS (EDGE architecture) [46] is designed with a dynamically routed mesh that connects the processing elements by the means of routers. The mesh connects each PE to 8NN with 4 inputs and 4 outputs. Different 64-bit buses connect each PE to tiled data cache resident beside the grid. A simple point-to-point links are used for control. The PE contains beside an ALU, registers for operands, router, control unit, frame configuration cache. The frame configuration cache can be of a capacitance up to 128 entries with instruction configurations and predicated data information. Hybrid dataflow are optimized with predicated execution techniques with special tools to form hyperblocks of configuration.

ADRES [6] architecture is a 2-D mesh hosted by a VLIW control processor. Each reconfigurable functional unit in this device contains a 32-bit ALU which can be configured to implement one of several functions including addition, multiplication and logic functions, with two small register files. To that, additional control elements and multiplexers are used to control the inputs/output and instruction selection. A configuration memory is attached to each ALU and features a multi-context configuration cache. The DRESC compiler aims at optimizing the program, where the mapping of the configurations to the grid occurs via interconnection mesh. Each PE is connected to all PEs on the same column and rows via horizontal and vertical interconnections.

PACT XPP processor [47] is partitioned to four tiles (PAC) to enable the mapping of different small basic blocks. Each PAC consists of a square of 64 32-bit ALU-based processing array elements (PAEs), 16 memory-based PAEs (one Kbyte each), and 4 32-bit I/O units (with two channels each) with an additionally adder/subtracter. A configuration manager is distributed on several abstract levels to map the configurations to the PAEs by the means of configuration buses. A local RAM is also distributed with the configuration manager. Each PAE contains an ALU as well as registers for vertical routing and horizontal routing. The PAC objects communicate via a packet-oriented network for data and events. In contrast to data packets, event packets are just one or a few bits wide, and they transmit 32-bit data for processing.

The RAW [48] [4] features a reconfigurable multicore architecture. The grid of RAW provides a mini-RISC processor as PE on the grid to conform a multi processor machine. The architecture is composed of NN connected 32-bit modified MIPS R2000 microprocessor tiles with ALU, 6-stage pipeline, floating point unit, controller, register file of 32 general purpose and 16 floating point registers, program counter, and local cached data memory and 32 Kilobyte SRAM instruction memory. The prototype chip features 16 tiles arranged in a 4 by 4 array. RAW provides both a static (determined at compile-time) and a dynamic routing (determined at run-time: wormhole routing for the data forwarding). Since the processors lack hardware for register renaming, dynamic instruction issuing or caching (like in superscalar processors), statically scheduled instruction streams are generated by the compiler, thus moving the responsibility for all dynamic issues to the development software. However, RAW provides possible flow control as a backup dynamic support, if the compiler should fail to find a static schedule.

KressArray [49] is primarily a mesh of reconfigurable datapath units (rDPUs) physically connected through wiring by abutment i.e. no extra routing areas needed. The KressArray is a super-systolic array (generalization of the systolic array). Its interconnect fabric distinguishes 3 physical levels: multiple unidirectional and/or bidirectional NN links, full length or segmented column or row backbuses, a single global bus reach-

ing all rDPUs (also for configuration). Each rDPU can serve for routing only, as an operator, or, an operator with extra routing paths. The 2nd and 3rd level is layouted over the cell. I/O data streams from and to the array can be transferred by global bus, array edge ports, or ports of other rDPUs (addressed individually by an address generator). The KressArray Family is supported by an application and development tool and platform architecture space explorer (PSE) environment. The basic principles of the KressArray define an entire family of KressArrays covering a wide but generic variety of interconnect resources and functional resources.

MATRIX [50] (Multiple ALU Architecture with Reconfigurable Interconnect eXperiment) is composed of multi-granular array of 8-bit basic functional units (BFUs) overlayed with a configurable network with procedurally programmable microprocessor core including ALU, multiplier, 256 word data and instruction memory and a controller which can generate local control signals from ALU output by a pattern matcher. Each functional unit contains a 256x 8-bit memory, an 8-bit ALU and multiply unit, and reduction control logic including a 20x8 NOR plane. The network is hierarchical, supporting three levels of interconnect. Functional unit port inputs and non-local network lines can be statically configured or dynamically switched.

REMARC (Reconfigurable Multimedia Array Coprocessor) [51], a reconfigurable accelerator tightly coupled to a MIPS-II RISC processor, consists of an 8 by 8 array of 16 bit nanoprocessors with 16-entry data memory and 16-bit registers attached to global control unit. The global control unit comprise 1024-entry instruction RAM controls the transferring of data between the host processor and nanoprocessors. The communication resources consist of NN direct connections and additional 32 bit horizontal and vertical buses which also allow broadcast to processors in the same row or column respectively, or, to broadcast a global program counter value each cycle to all nanoprocessors, also to support SIMD operations.

The CHESS [41], a hexagonal array of alternating ALU/switchbox sequences. Chess

array features a chessboard-like floorplan, where each ALU is adjacent to four switchboxes. Switchboxes can be converted to 16 word by 4 bit RAMs if needed. To avoid routing congestion, the array features also embedded 256 word by 8 bit block RAMs. An ALU data output may feed the configuration input of another ALU, so that its functionality can be changed on a cycle-per cycle basis at runtime without uploading. However, partial configuration by uploading is not possible. The interconnect fabrics of CHESS has segmented four bit buses of different length. There are 16 buses in each row and column, 4 buses for local connections spanning one switchbox, 4 buses of length 2, and 2 buses of length 4, 8 and 16 respectively.

DReAM Array (Dynamically Reconfigurable Architecture for Mobile Systems [52]) consists of 2-D array of parallel operating coarse-grained reconfigurable processing units (RPU). Each RPU consists of: 2 dynamically reconfigurable 8-bit reconfigurable arithmetic processing (RAP) units, 2 barrel shifters, a controller, two 16 by 8-bit dual-ported RAMs (used as LUT or FIFO), and a communication protocol controller. The RPU array fabric uses NN ports and global buses segmentable by switching boxes.

RaPiD (The Reconfigurable Pipelined Datapath (RaPiD) [53]) is a linear array of arithmetic-oriented units, including 15 DPUs of 8 bit with integer multiplier (32 bit output), 3 integer ALUs, 6 general-purpose datapath registers and 3 local 32 word memories, all 16 bits wide. It aims to speed-up of highly regular, computation-intensive tasks by deep pipelines on its 1-D grid. Each memory has a special datapath register with an incrementing feedback path. To implement I/O streams, RaPiD includes a stream generator with address generators, optimized for nested loop structures, associated with FIFOs. The address sequences for the generators are determined at compile-time. RaPiD's routing and configuration architecture consists of several parallel segmented 16 bit buses, which span the whole array. The length of the bus segments varies by tracks. In some tracks, adjacent bus segments can be merged.

PipeRench [54], an accelerator for pipelined applications, provides several reconfig-

urable pipeline stages (stripes) and relies on fast partial dynamic pipeline reconfiguration and run time scheduling of configuration streams and data streams. It has a 256 by 1024 bit configuration memory, a state memory (used to save the current register contents of a stripe), an address translation table (ATT), four data controllers, a memory bus controller and a configuration controller. The reconfigurable fabric of the PipeRench allows the configuration of a pipeline stage in every cycle, while concurrently executing all other stages. The fabric consists of several (horizontal) stripes composed of interconnect and PEs with registers and ALUs, implemented as 3-input lookup tables. The ALU includes a barrel shifter, carry chain circuitry, etc. A stripe provides 32 ALUs with 4 bits each. The whole fabric has 28 stripes. The interconnect scheme of PipeRench features local interconnect inside a stripe as well as local and global interconnect between stripes and four global buses.

The Pleiades Architecture [55] is a generalized low power extension of previous versions, PADDI-1 [43] and PADDI-2 [44] with programmable microprocessor and heterogeneous grid of execution units (EXUs). It allows to integrate both fine and coarse grained EXUs, and, memories in place of EXUs. For each algorithm domain (communication, speech coding, video coding), an architecture instance can be created (with known EXU types and numbers). Communication between EXUs is dataflow driven. The control means available to the programmer are basic EXU configurations to specify its operation, and interconnect configurations to build EXU clusters. All configuration registers are part of the processor's memory map and configuration codes are processor's memory writes.

Chameleon Systems offers a reconfigurable platform for telecommunications and data communications, with a 32 bit RISC core as a host, connected to a RA fabric with 108 DPUs (84 32-bit ALUs and 24 16-bit multipliers), arranged as 4 slices by 3 tiles a 7 ALUs and 2 multipliers each, including an 8 word instruction memory for each DPU and 8 Kbytes of local memory for each slice.

Architecture	Grid	Processing Element	CF/data Granularity	Connectivity
MorphoSys	4 tiled 2-D mesh	32-bit alu,mul,shi,RF,2mux	16 bit	NN, length 2 & 3 global lines
TRIPS	2-D array	32-bit alu,Op-Re,router,CU,CM	32 bit	8NN, 4 64-bit global lines
ADRES	2-D array	32-bit alu,RF,3-mux,CU,CM	32 bit	vertical/horizontal, 32-bit bus
PACT-XPP	4 tiled 2-D array	32-bit alu, adder,router,CM,CU	32 bit, multi-granular	horizontal roting
RAW	2-D mesh	Mini-RISC,router	8 bit, multi-granular	8NN switched connections
KressArray	2-D mesh	rDPU	family: select pathwidth	multiple NN & bus segments
RaPID	1-D array	rDPU	16 bit	segmented buses
Matrix	2-D mesh	8-bit alu,CU,CM,memory-block	8 bit, multi-granular	8NN, length 4 & global lines
Pleiades	mesh/crossbar	16-bit alu,mul,switch box,CM	multi-granular	multiple segmented crossbar
PipeRench	1-D array	4-bit alu,shi,LUT	128 bit	(sophisticated)
REMARC	2-D mesh	16 bit nanoprocessors	16 bit	NN & full length buses
CHESS	hexagon mesh	alu/switchbox	4 bit, multi-granular	8NN and buses
DReAM	2-D array	8-bit ALU,2-shi,CU,CM	8 &16 bit	NN, segmented buses
Chameleon	2-D array	DPU,32-bit alu,16-bit mul	32 bit	co-compilation

Table 2.1: coarse-grained reconfigurable architectures, where Op-Re: operand registers, RF: register file, mul: multiplier, shi: shifter, mux: multiplexers, CU: control unit, CM: configuration memory, rDPU: reconfigurable data path unit, NN: nearest neighbour,

Architecture	Mapping	application domain
MorphoSys	manual P&R	(not disclosed)
TRIPS	routing	general purpose
ADRES	DRESC compiler	general purpose
PACT-XPP	Configuration manager	image processing
RAW	switch box rout	experimental
KressArray	(co-)compilation	(adaptable)
RaPID	channel routing	pipelining
Matrix	multi-length	general purpose
Pleiades	switchbox routing	multimedia
PipeRench	scheduling	pipelining
REMARC		multimedia
CHESS	JHDL compilation	multimedia
DReAM	co-compilation	next generation wireless
Chameleon	(not disclosed)	tele- & datacommunication

Conclusion: Although there are many benefits of coarse grain architectures, many hurdles reduce their efficiency. Time and effort are needed for the high-level software-tool design that applies optimizations to the program to be executed. Additionally, the cooperation between the hardware and software and the time needed for mapping the configurations on the bus to grid has reduced the expectations of a much better performance. These architectures suffer also from adapting the design to execute out-of-order and enabling data-cache/memory accesses. Coarse grain designs have inherited the memory bottleneck from general purpose microprocessors. To that, a misprediction during the execution can result in a high latency, since a new configuration must be mapped. The connectivity of the PE is another issue to be tuned, since it directly affects the performance, the hardware costs and the power consumption. The most of coarse grain reconfigurable designs suffer from low hardware utilization, because of the poor connectivity and high interconnection latency problems. Beside the interconnection, the routers apply another latency factor, which increases the delay of result deliverability.

2.4 Toward Hybrid Architectural Design

All presented technologies bear many advantages and efficiency factors. However, a more scalability of each technologies faces many hurdles as already discussed. The limitations of the presented technologies can not be broken by traditional ways of performance enhancement. However, new hybrid ideas can achieve a break-through in the underlaying technology limitations. Based on this observation, we offer in this research a mixed design that combines several efficiency factors of the architectural design of these technologies. By this way, we introduce a new path for accelerating the execution of application beyond that of out-of-order superscalar processors, more dynamic execution than similar coarse grain architectures, and an efficient asynchronous execution applied only to the long data path in the design.

Thus, the GAP is taking the advantages and avoiding the bottlenecks of these technologies by:

- Moving the configuration task from the software to the hardware keeping the advantage of high ILP and loop acceleration in the grid. Avoiding herewith the software design stage and the analysis of the application, that must be done in advance to the execution.

- Implementing a superscalar frontend avoiding the issue bottleneck by allowing the simultaneous issue/configuration of dependent and independent instructions.

- Taking the benefit of GALS designs by allowing asynchronous execution only inside the long data path (the grid). This accelerates the execution of the instruction without complex synchronization mechanisms.

Chapter 3

The Grid Alu Processor

3.1 Architectural Overview

The basic idea of the GAP architecture is to combine different efficient characteristics of the three basic technologies, superscalarity, asynchronous execution, and coarse grain reconfigurability. Concurrently, It avoids the hurdles and bottlenecks—mentioned in previous chapter 2—of these design methodologies.

The GAP architecture combines a superscalar processor frontend and a data-driven-alike execution core fully integrated to a single core design. Unlike similar architectures which extract the configurations in the software, the GAP is able to extract the configurations in the processor frontend and map them to the execution core. Extracting the configurations in the hardware keeps the processor transparent to the software and does not imply any compiler modifications. In contrast, coupling an accelerator with a controlling processor demands a lot of analysis in the software and special mapping mechanism to the hardware. Data-driven-alike execution cores allow a fast execution of special code snippets like loops. However, the dynamic hardware-based reconfiguration and the mapping of instructions at runtime into an execution core requires a

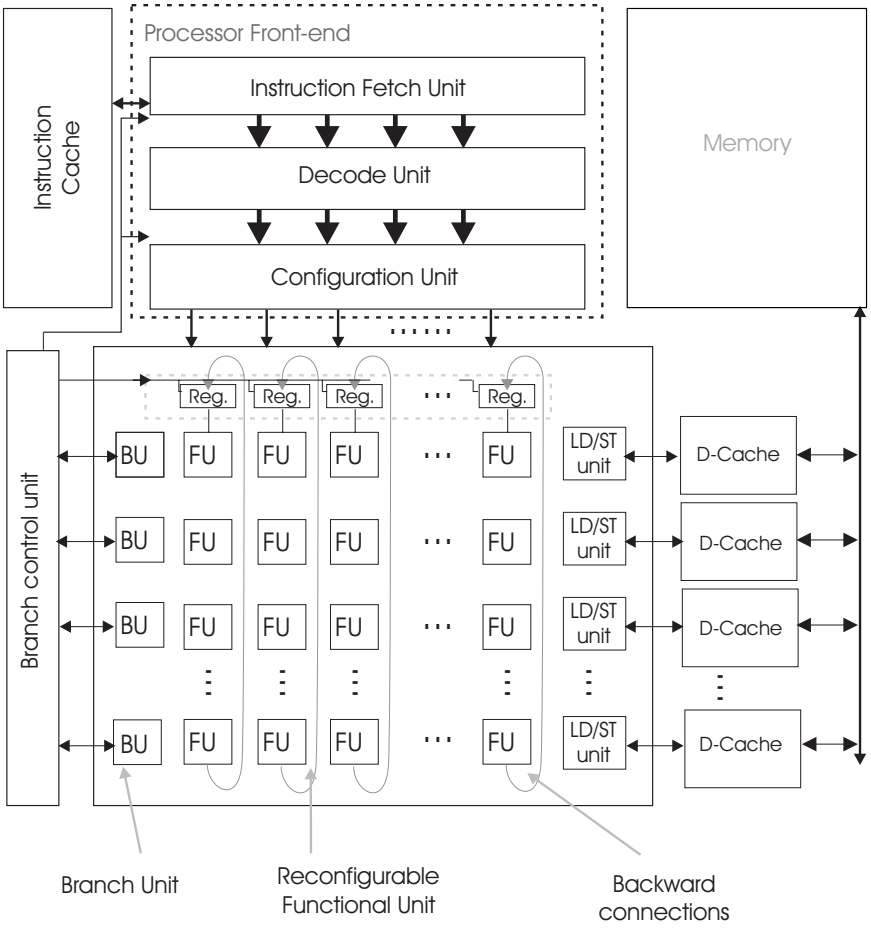


Figure 3.1: GAP Architecture

special hardware unit, we called configuration unit as shown in Figure 3.1. Thus the frontend and the execution core are combined by a configuration unit that dynamically maps a conventional sequential instruction stream to the FUs inside the execution core. The configuration unit is responsible for the mapping of maximally four instructions each clock cycle to the execution core. This configuration task demands the ability of mapping dependent and independent instructions simultaneously, since the instructions belong to a conventional sequential instructions stream.

The execution core is arranged as a two dimensional array of FUs and executes asynchronously. Together with some additional components for branch execution and mem-

ory accesses (at the right side and left side of the array respectively) the GAP is able to execute conventional program binaries. The array can mainly accelerate the execution of loops by saving their configurations inside the grid and execute them asynchronously exploiting herewith a high ILP. Other code structures can also benefit from the grid by simultaneously mapping of dependent and independent instructions and executing them asynchronously. This accelerates the execution of critical paths in sequential applications. In contrast, an out-of-order processor suffers latencies from dependent instructions as they wait in the issue buffer to be waked up after finishing the preceding instructions. Dependent instructions can occupy the issue buffer for long times especially when they depend on a memory access instruction.

The hardware reconfigurability allows the deployment of an in-order processor frontend avoiding the large, unscalable hardware structures of out-of-order processors, like: large issue windows and the needed hardware to control it, renaming structures, and reorder buffer. Nevertheless, the GAP reaches the throughput of an out-of-order processor by simultaneously mapping dependent and independent instructions. The execution inside the array is out-of-order and similar to data-driven architectures. Each FU starts the execution when the values of both sources are available. However, this execution scheme requires a special synchronization to deliver the correct results to the synchronous components around the array. Instruction execution with timing scheme is described in more detail in Section 3.3.2.

The array is arranged in rows and columns of FUs (see Figure 3.1) that are described in Section 3.3 in details. Each column in the basic GAP architecture is accompanied by an architectural register of the processor's instructions set. We choose the Portable Instruction Set Architecture (PISA) known from the SimpleScalar simulation tool set [56]. The PISA instruction set architecture is designed with 32 physical registers. Hence, the basic array contains 32 columns for the general purpose registers and two additional columns for the multiply/divide registers (hi/lo) with the corresponding functional units (one multiply/divide unit per row). The set of the registers at the top of the columns

conforms a register file. All registers must be read when the execution starts in the array, as this registers contain the starting value of the corresponding architectural register. Result write back to these registers occurs when the execution inside the array has been finished. The write back mechanism is simple in this case, since each column writes to the accompanied register. However, the fixed number of columns features a non efficient factor of the hardware costs of the array. Section 4.3.1 is devoted for the solution of this problem without to change the data-flow-execution nature.

The general data flow within the array takes place in a top-bottom wise. Each FU is able to read the output values of every FU from the previous row as inputs and its output is available for all FUs in the next row. Thus, there is no data exchange within the same row and the results can not be bypassed to the rows up. In general, each FU is configured by a machine instruction or it is configured as route forward, i.e. the FU bypasses the data from the previous FU in the same column to the next one. Bypassing the result by the FU that does not bear an instruction makes the result available not only for the FUs on the next row but also to all FUs on the down-lying rows.

Each row in the array is accompanied by a memory access unit that serves as communication interface to the data cache. Memory access units read the address/data from the previous row and send the request to the accompanied data cache. The read value from the cache forwards to the consuming FU on the next row in the case of a load access. In the store case, the access finishes after sending the address and the data to the cache.

The branch control unit is equipped with a connection to each branch units (BU) (shown on the left side of the array). Each BU can hold the configuration of a branch instruction. A false estimation of the branch condition lets the execution of the following instructions in the array proceed as usual. However, a true estimation of the branch condition signals a finished execution in the array. Then, the branch control unit sends the new address to the program counter in the fetch unit. Simultaneously, it activates the result write back to the top registers of the array.

Code Execution Example:

The placement of instructions into the array is demonstrated by the following simple code fragment of pseudo machine instructions that adds fifteen numbers out of subsequent memory cells followed by negating the result.

```
1:    move R1,#15     ;15 values
2:    move R2,#addr   ;start address
3:    move R3,#0      ;Register for the sum loop:
4:    load R4,[R2]    ;load an element
5:    add R3,R3,R4    ;add
6:    add R2,R2,#4    ;inc address
7:    sub R1,R1,#1    ;dec counter
8:    jmpnz R1,loop   ;end of loop?
9:    sub R3,R1,R3    ;negate the result (R1=0)
```

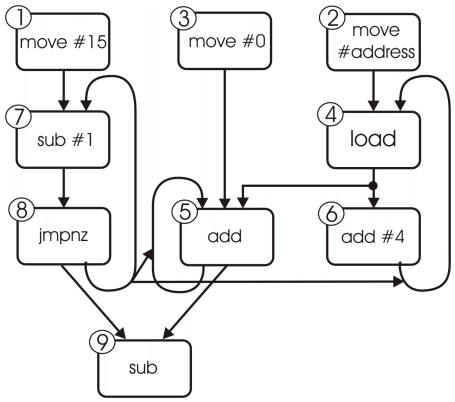


Figure 3.2: Dependency graph of the example instructions.

Figure 3.2 shows the dependency graph of the 9 instructions, which can be recognized again at the placement of the instructions within the GAP backend shown in Figure 3.3. Supposing that no miss in the instruction cache occurs during the fetch of these instructions, the configuration unit maps four instructions each clock cycle to the array. In

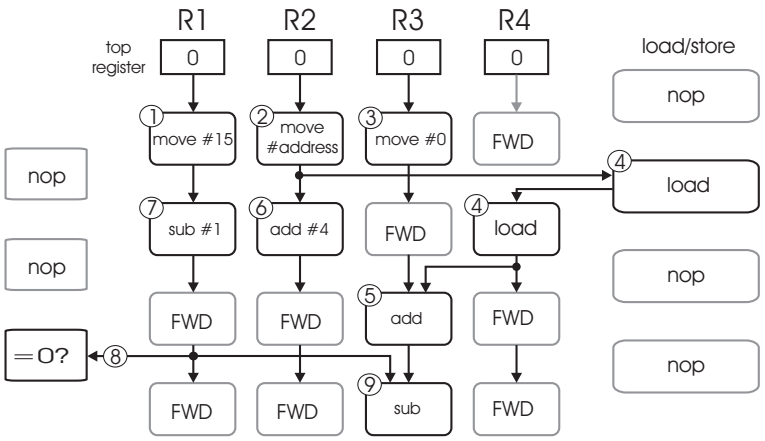


Figure 3.3: Placement of the complete example fragment

clock cycle *x* instructions 1 to 4 are placed in the corresponding FUs. The instructions 1 to 3 are placed within the first row of the array. Instruction 4 depends on $R2$ which is written in the first row and, therefore, it must be located in the second row. It reads the address as a result of instruction 2 and forwards the data received from memory into the column $R4$ which is the destination register of the load. Hence, data dependent instructions are placed in different rows regarding the network structures deployed in the grid, where the network interconnections allow the results to flow in top-bottom wise as stated previously. In the clock cycle *x+1*, the instructions 5 to 8 are placed in an analog way. Instruction 8 is a conditional branch that could change the program flow. To sustain the hardware simplicity, conditional branches are placed below the lowest already arranged instruction. In this case, the branch has to be located after the third row. Therefore, the last instruction is placed below the branch in the fourth row. All other FUs that are not configured by an instruction stay in forwarding status (FWD) to bypass the result of the previous FU on the same column to the next one.

During the mapping of the configurations of instructions 5 to 8, the execution of the previous four instructions starts. Each instruction starts its execution when both operands are available. Estimating the branch to be taken, the result of the last instruction must be discarded to enable forwarding the correct calculated results and write

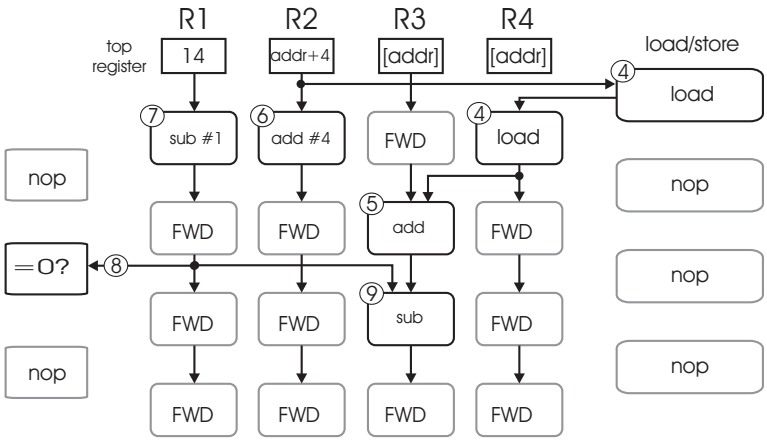


Figure 3.4: Placement of the loop body (instructions 4 to 8) and the subsequent instruction (instruction 9)

them back to the top registers. Otherwise, the result of the instruction 9 must be considered to be written back to the corresponding register $R3$. Thus, not matter whether the instruction 9 is executed before the branch instruction or later, the correct execution results will be written back to the registers at the end of the configuration phase. Hence, the execution inside the array is completely out-of-order within both basic as well as control-based code structures mapped to the array.

In this example, the *loop* address points to the instruction 4, where instructions 4 to 8 conform a loop body. In the case of a taken branch, a jump to the instruction 4 must be performed. The branch target is already mapped into the array but it is placed among other instructions that must not be executed again. Therefore, the instructions 4 to 9 are fetched and mapped again into the array as shown in Figure 3.4. Now, the loop target is the first instruction within the array, and hence, all subsequent loop iterations can be executed directly within the array without any additional fetch, decoding and configuration phases. The mapping of new instructions that follow the loop body continues until the array is full. After a full array is signaled, the processor frontend stalls waiting the array to finish executing the instructions already mapped. Due to the placement of

instructions following the loop body, the GAP does not suffer from any misprediction penalty at loop termination as other processors using branch prediction would do. Additionally, many instructions outside the loop will be ready to be executed, which exploits a high ILP on the first hand, and shortens the execution of the critical path of the mapped code by the asynchronous execution on the second hand.

3.2 Processor Frontend

The fetch unit, the decode unit, and the interface to the instruction cache are similar to those of superscalar processors. To this, the hardware reconfiguration allows the deployment of in-order, simple frontend in the processor avoiding the issue bottleneck of superscalar architectures. The fetch unit is able to fetch four instructions each clock cycle out of a normal program binary generated by a standard compiler. A program counter determines the position of the next instructions to be fetched.

The configuration of each instruction is extracted by simply decoding the instructions to their operands. The decoded instructions proceed to the configuration unit, where the mapping decision takes place. The configuration unit is able to map dependent and independent instructions of a conventional instruction stream into the array of FUs. Thus, the in-order processing in the frontend does not feature a disadvantage to the throughput, as the execution in the array is out-of-order and asynchronous as well. Each instruction starts its execution on the bearing FU when the input values are valid.

The processor frontend and the reconfigurable fabric operate in parallel. Hence, the execution of each instruction starts directly after mapping it to the corresponding FU. Thus, The frontend of the processor stay active as long as the array is not full. The frontend continues mapping dependent and independent instructions to the array unless a taken branch is detected by the branch control unit. By a taken branch the fetch unit receives the outstanding address from the branch controller and adjusts the program

counter accordingly. If no taken branch is detected, the mapping of instructions continues until the array becomes full. After filling the array, the frontend stalls and the backend keep busy with executing already placed instructions in the array. This occurs when some long latency operations like load instructions have to be executed or if a loop is executed. In both cases, the frontend can switch into sleep mode to save energy.

3.2.1 Configuration Unit

As mentioned previously all related architectures move the burden of the reconfiguration on the software side or the compiler. The software algorithms can easily be modified to be aware of the underlying hardware structures and interconnections on the grid. Depending on the hardware geometries and the instruction dependencies, the software can make a decision where to map each instruction. However, the cooperation between the hardware and the software in this case reduces the gain of performance resulting from loop acceleration. Especially with the architectures that allow to execute control change instructions inside the grid. A misprediction can incur a penalty time when the new configurations must be fetched from the memory to the configuration cache (when available) and then to the grid.

Our solution methodology however is based completely on the hardware to achieve more adaptivity and dynamicity allowing herewith the deployment of common compilers without any modifications. A novel configuration unit fully integrated into the pipeline allows the resource aware effective mapping of instructions with simple hardware requirements in the grid.

The reconfiguration takes place at runtime. Principally, the output of the decode stage constitutes the configurations of the instructions to be mapped. Consequently, the configuration unit maps these configurations each clock cycle into the corresponding FUs in the array. The mapping task of the configuration unit must take into account

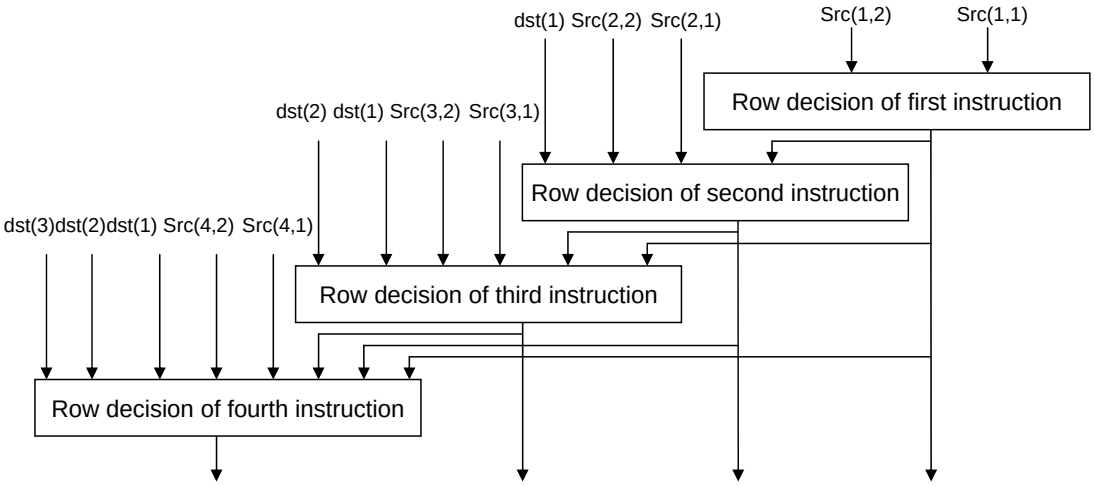


Figure 3.5: Block diagram of the row decision

the availability of the underlying hardware structures and the data dependencies. In our basic design, we assume a network between the functional units designed in a way that two adjacent rows are completely connected. Every FU in the array can read the results of the FUs in the previous row and forward it (or its own result) to all FUs in the next row. Thus, instructions with data dependencies have to be mapped to different rows where the result deliverability is ensured by the available interconnections.

The mapping task of the configuration unit is responsible only for row selection depending on the data dependencies. Column selection is simplified by assigning each column in the array to a specific destination register (physical register). With this restriction, we allow a simple input selection for each FU by using both sources of the instruction as configurations. Each source directly controls a multiplexer to select the input coming from a specific column i.e. the specified source value. Thus, the clear assignment of instructions to specific columns allows a very simple input selection without the need for routers or attaching a register file to each FU. This mapping strategy leads however to an array with a fix number of columns, which is ineffective regarding hardware costs. The optimization of the number of columns is discussed by an example in

Section 4.3.1.

The row selection of the four instructions to be mapped in one clock cycle is a dependent decision as shown in Figure 3.5. A correct decision requires a comparison of both sources of each instruction with the destinations of all previous instructions to be mapped in the same clock cycle. Hence, a data dependency between two instructions must be considered to map the instructions to different rows, where the second instruction can read the result of first one.

Both data dependencies: true-dependency and anti-dependency are considered in the configuration stage. The true-dependency is the case when second instruction read the result of first instruction as an input as shown on the left side of Figure 3.6, where A and B:

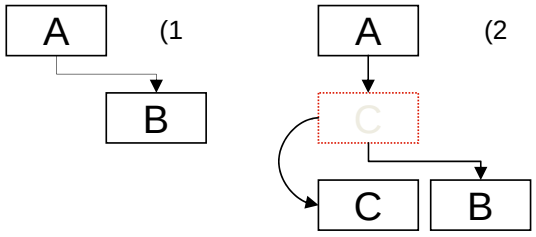


Figure 3.6: 1) true-dependency, 2) anti-dependency

```
A: add dst:r3, src1:r2, src2:r5
B: add dst:r6, src1:r3, src2:r7
```

The anti-dependency is the case when a third instruction try to write the output of first instruction before it read by second one. In the case of anti-dependency the third instruction is drop down to the same of the second instruction to enable reading the correct result of instruction A, where A, B, and C shown on the right side of Figure 3.6:

```
A: add dst:r3, src1:r2, src2:r5
B: add dst:r6, src1:r3, src2:r7
C: add dst:r3, scr1:r3, src2:r8
```

The described configuration unit with the functionality of mapping the instructions according to their data dependencies into the array of FUs requires few information about the underlying hardware structures. The required information is only the number

of available rows and in which row each register value is available. Using status registers (as shown in Figure 3.7) to save and update the rows in which the values of the registers are available, makes the mapping task feasible. As shown in the figure, the row decision of first instruction is simple and requires only a comparison of the status register of both sources (St(1,1), St(1,2)). The most significant bit (MSB) of the comparison is used to control a multiplexer to choose the row on which both sources are available. The decision on following instructions must compare the destination of all previous instructions with both sources of each instruction, as a true-dependency could exist. If one of the sources is equal to the destination register of a previous instruction, the row decision of current instruction is made easily by taking the row decision of the previous instruction and increasing it by a one. Moreover, the both sources of all previous instructions is compared with the current destination to avoid anti-dependence case. In this case, the row decision is the same as of the previous instruction to which anti-dependency exists.

The complexity and the propagation delay of this circuit grows with the number of instructions to be mapped each clock cycle. The anti-dependency and true-dependency check calculations does not lie on the critical path as shown in the figure. However, the number of implemented rows in the array impacts the number of status bits of each register. As an example, an array with 4 rows requires status registers with only two bits. With more bits (i.e. more rows) for each status register, the size of the operational component (the comparators (CMP) and the multiplexers (MUX)) in the circuit grows.

After making the decision of row mapping, the status registers must be updated. Each status register related to a destination register or source register of the mapped instructions is then updated with the same value as the accompanied row decision value. The updating of a status register when the accompanied register is used as destination is obvious, since the value for next instructions will be available starting only from current row decision. Also, if the reference register is used as source register, the status register must be updated with the row decision value. This must be done in order not to allow writing the source register by another following instruction before the source is read

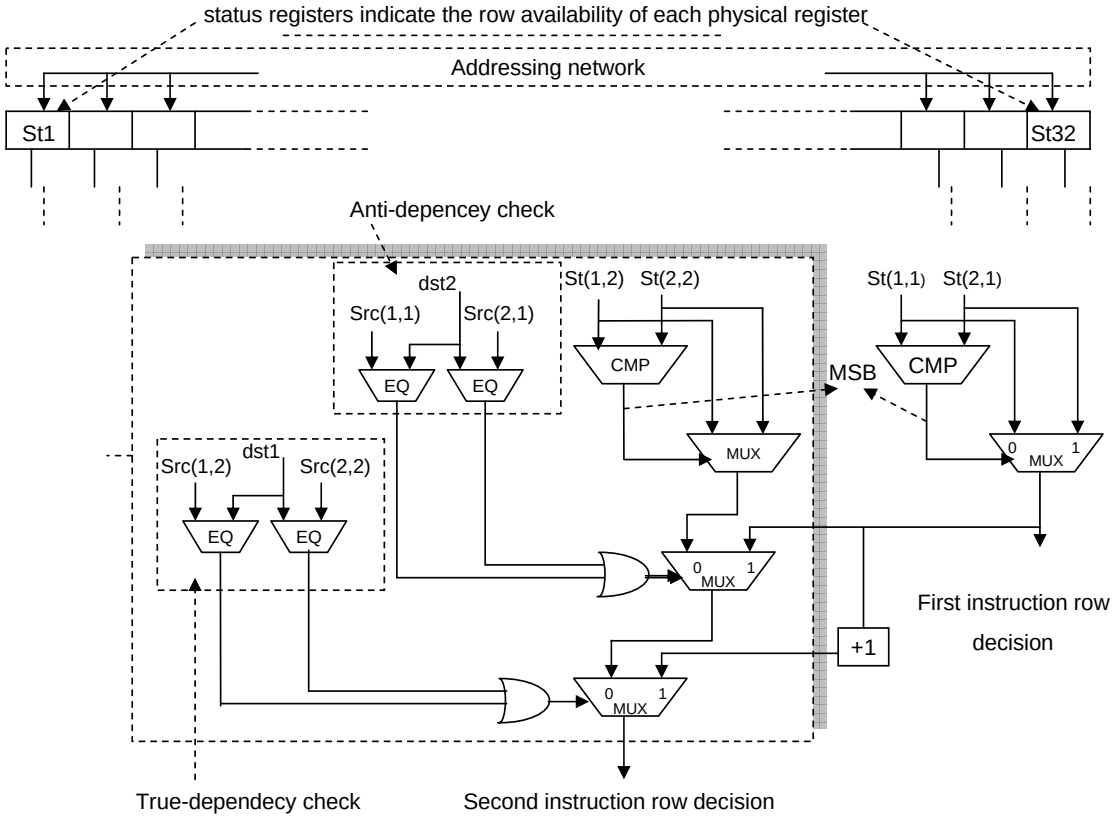


Figure 3.7: The row selection circuitry of two instructions in the configuration unit

(anti-dependence case). The status bits have to be reset each time the configurations in the array are deleted due to a taken branch (a new configuration phase). Also, if the row decision of an instruction is greater than the number of available rows, the configuration unit signals a full array and stalls the frontend until the execution in the array is finished. A full array can not necessarily be detected with the first instruction and it does not make a sense to withdraw the decision of previous instructions in the same clock cycle that have successfully completed their decisions. Thus, the four instructions are not considered as complete packet of configurations. However, if an instruction signals a full array, all following instructions in the same clock cycle stop the row selection even when its possible to map one of them to the array. The mapping of the rest of instructions starts only after finishing the execution in the array.

Operation	Pico-cycles
Add, Sub	3
Stlt, Stgt	3
And, Or, Xor, Not	1
Shl, Shr	2

Table 3.1: Pico-cyle times assumed for the evaluation

A full array signal resets the status registers in the configuration unit. The reconfiguration starts again after finishing the execution in the array as already explained. Also with a taken branch in the array, the branch control unit signals a new configuration phase and leads to reset the status registers in the configuration unit. The mapping of the new coming instructions starts from the top of the array.

3.2.2 Timing Analysis in the Frontend

As mentioned previously, the execution of the operations inside the array is asynchronous. Therefore, the time at which the valid calculated results arrive the boundaries of the array (when finishing the execution, acquiring data from the memory or when a branch is taken) is not known. Hence, it is necessary to set a timing scheme that controls the interface communications between synchronous and asynchronous components. Therefore, beside the placement of operations into the array, the frontend is also responsible for the timing inside the grid. To synchronize with the synchronous components around the array, the processor frontend is aware of the timing of each operation. The runtime of each operation is known in the processor frontend in terms of so-called *pico-cycles*. We have chosen one pico-cycle as $\frac{1}{4}$ of a machine clock cycle.

The timing scheme is set up according to the propagation delay of each operation.

As shown in Table 3.1, we assigned each arithmetic logic operation to a propagation delay estimation (the real propagation delay of each operation can only be known in the stage of the ASIC design). Based on this table the duration of each operation is known in the processor frontend. With the help of a counter for each column in the array, the time at which a valid result arrives one of the boundaries is calculable. Each time an instruction is decoded, the propagation delay of the current operation plus the critical path of both sources is added to the counter of the destination register. Based on these counters, a timing scheme control is set to ensure a correct communication between the asynchronous and synchronous interfaces. The detailed description of the timing scheme inside the array and the synchronization is demonstrated in Section 3.3.2.

The necessary information for the timing scheme is the table of the propagation delay and the operands of the instructions (OP-code and both source). It is also obvious that the calculations for the timing scheme are completely independent of the row decision in the configuration unit. Hence, these calculations do not lie on the critical path of the configuration unit. It can also be done earlier in the decode stage or in a separate stage after the instruction decoding. This decision can be solely done after the analysis of the critical paths in the pipelining phase of the design.

3.3 Processor Backend

The GAP backend comprises the FU array, the branch controller, and the memory access units. The FUs are connected by an interconnection network to enable the result deliverability between the data dependent instructions. The arrangement of FUs, load/store units, and branch controller as shown in Figure 3.8 allows the mapping of data flow as well as control flow oriented parts of the program without any change of the generated binary code of a conventional compiler.

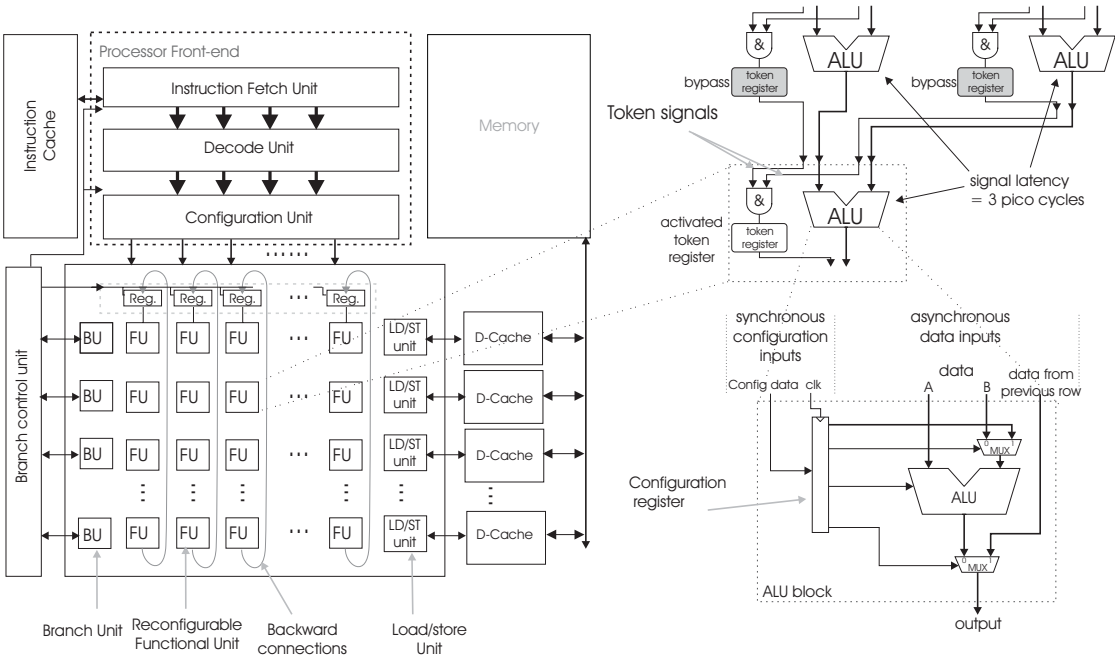


Figure 3.8: GAP Architecture

3.3.1 Hardware Structures in the Grid

The backend comprises a two dimensional array of identical FUs that are able to perform simple arithmetic logic operations. These operations are: *add, sub, shl, shr, and, or, xor, not, neg, stlt, stgt*. For complex operations, a special column is dedicated with one multiplier/divider in each row (the hardware optimization of the complex functional units is illustrated in Section 4.2). Branch units as well as load/store units are similar to the FUs in the array as they perform similar operations (addition/subtraction). Each FU contains an ALU accompanied by a configuration register, three multiplexers for input/output selection, and a timing circuitry for controlling the token signal as shown in Figure 3.9. The token signal flows in parallel to the data calculation and serves as synchronization with the synchronous components around the array. The timing scheme explained previously must program this token signal to arrive simultaneously with the

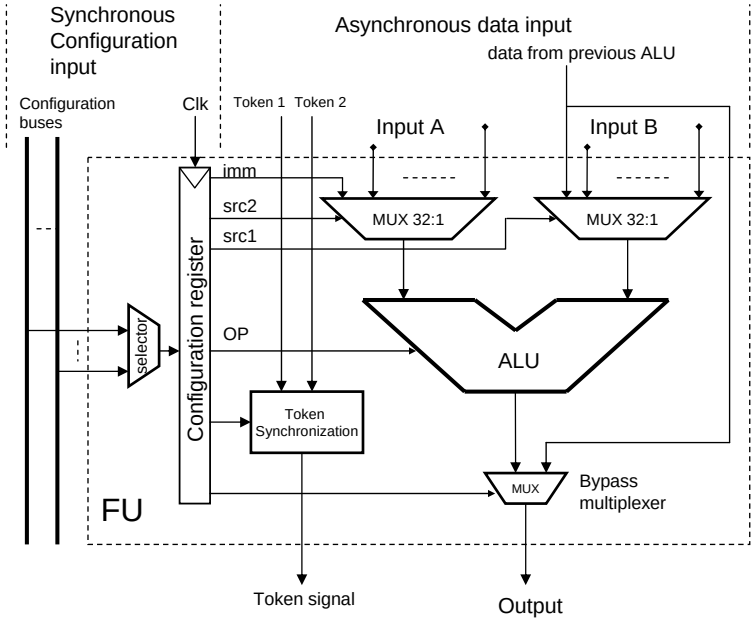


Figure 3.9: Block diagram of a functional unit comprising an ALU, input selectors, bypass network, and a configuration register set by one of two configuration busses

valid results at the boundaries of the array.

Each ALU is accompanied by a configuration register constitutes of a number of bits representing the OP-code, first source register, second source register, and the optional immediate value. The multiplexer at the output selects the calculated result of the FU (when the FU is configured with an instruction) or the result of the previous FU on the same column (when the FU is configured to bypass).

Since at most four instructions must be configured each clock cycle, four configuration buses with each column propagate from the configuration unit to end of the array. After row selection in the configuration stage, the configuration data as well as the address of the FU are sent on one of the dedicated four buses. With the help of selector, each FU can recognize when it is addressed by the configurations on the bus. However, these buses get longer when increasing the number of rows in the array. Nevertheless,

on the first hand this does not introduce a technical obstacle to the hardware designer, since these wires can be pipelined to scale with the clock cycle. On the second hand instructions that must be mapped to deep rows are not critical to the execution time, since the execution is mostly delayed by memory access instructions in the top of the array. Another design possibility can be considered by deploying only two configuration buses beside each column in the array. In this case, only two instructions with the same destination register can be placed to the array in single clock cycle. By configuring more than two instructions with the same destination register, the processor frontend stalls for one clock cycle to complete mapping the rest of instructions.

The execution of the operations starts directly after the mapping of new configurations to the array, since the processor frontend and the array are executing in parallel. However, the execution inside the array takes place with asynchronous timing, whereas the frontend (including the configuration unit) maps the instructions synchronously. Hence, the deployment of registers to save the values temporarily at the input or the output of each FU is unnecessary in our design. To this, each column is associated with a specific destination register, which allows using multiplexers for the input selection. Moreover, the simple asynchronous timing with simple input selection saves the need for extra communication components like routers.

3.3.2 The Timing Inside the Grid

The data execution starts at the top of the array at the begin of each configuration phase. Data calculation flows in parallel to a special token signal (shown in Figure 3.10) in order to synchronize with the synchronous units around the array. If the token signal arrives at a load/store unit, the request will be send to the D-cache in the next rising edge of the clock cycle. Similarly, a taken branch will be considered by the branch control unit only when the token signal arrives at the branch unit. Writing back the results at the end of each configuration phase is also controlled by the token signal

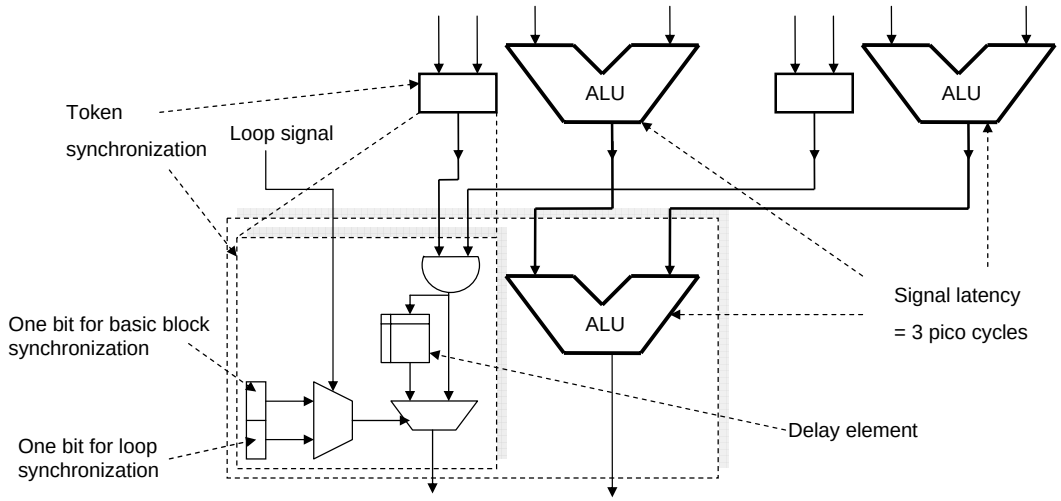


Figure 3.10: Three instructions, each requires 3 pico-cycles. The token register of the upper FUs are bypassed and the one of the lower FU is activated

to ensure writing the valid results. The penalty for synchronizing with synchronous structures (data caches, fetch unit) around the array is only a fraction of a clock cycle. Consequently, the resulting penalty yields a negligible wasted time in comparison to the time saved by the asynchronous execution.

The timing scheme in the processor frontend controls the token signal by analyzing the critical path calculation as briefly explained in Section 3.2.2. The frontend activates a token register to hold the token signal for one clock cycle if the calculation of the data on the critical path takes more than a clock cycle. This is done to ensure the simultaneous arrive of token signal and valid results at synchronous components. For an example, the token register of the framed ALU on the Figure 3.10 is activated under the assumption that 1 clock cycle is equal to 4 pico cycles (where the calculation in each ALU takes 3 pico cycles). Thus, during the configuration of each instruction, the delay time (i.e. calculation time) on each path of the sources must be considered. In the case where the critical path of the both sources plus the duration of the current operation is more than a clock cycle, the token signal is hold for one clock cycle. To hold the token

signal for a clock cycle, a single bit in the configuration register is set to choose the output of the delayed token signal as shown in Figure 3.9. In general, this bit is set to zero to bypass the token signal when the calculation on both paths is less or equal a clock cycle.

Basically, the mapping of the instructions takes place in parallel to the execution in the grid. Thus, the registers that contain the delay time of each physical register must be updated each clock cycle. For example, at clock cycle *x* an *add* operation is mapped to a FU in the array. At clock cycle *x+1* another *add* operation is mapped to the same column. In this case, the token signal must not be hold with the second instruction, since the execution of the first instruction is already done during the mapping of second instruction, even the theoretical critical path exceed the clock cycle. Hence, the registers that hold the delay time of each physical register must be subtracted by 4 each clock cycle (as each clock cycle is equal to 4 pico cycles and the delay time is calculated in terms of pico cycles). However, if both mentioned instructions belong to a loop body, then, the token signal must be hold, since in loop mode the instructions are already mapped. Hence the critical path calculation for loops differs from basic block calculation. Based on this timing scheme, two synchronization bits are used to separate between the dynamic mapping-execution and the loop-mode timing as shown in Figure 3.10. The first mapping of a loop belongs also to basic block synchronization, where the loop mode execution is the case where a loop has been captured and detected inside the array. In the case of loop execution, a global signal (loop signal) is activated to select the appropriate synchronization bit.

The following formula explains the tasks of the timing scheme in the frontend during the mapping of instructions. The instructions can be expressed in terms of the operation, both sources, and the destination register $inst(OP, src1, src2, dst)$ and the delay of each physical register is: $Delay(r(i))$. Thus each clock cycle the following step must be done in dynamic mapping-execution timing regardless whether instructions are available to be mapped or not: $\forall i \in \Psi : delay(r(i)) = |delay(r(i)) - 4|$, where Ψ is

the set of the physical registers. For loop-mode timing this step is not required, where all following steps have to be done for both timing bits.

In the case where the instructions are ready to be mapped, then the following timing analysis is performed. For each instruction a critical path comparison of both sources D_{cp} is done as in the following:

$$D_{cp} = \begin{cases} delay(src1) > delay(src2) & \text{then } delay(src1) \\ delay(src1) <= delay(src2) & \text{then } delay(src2) \end{cases}$$

Now, the critical path of the destination register is: $Delay(dst) = D_{cp} + D_{op}$, where D_{op} is the propagation delay of the operation.

After finishing the analysis, the holding decision of the token signal H_{token} is made as following:

$$H_{token} = \begin{cases} 1 & \text{if } delay(dst) > 4 \\ 0 & \text{otherwise} \end{cases}$$

Holding the token signal results in setting the synchronization bit in the functional unit. However, this requires updating the delay of the destination register.

$$delay(dst) = \begin{cases} |delay(dst) - 4| & \text{if } H_{token} = 1 \\ delay(dst) & \text{if } H_{token} = 0 \end{cases}$$

This delay is then saved in the delay register of the corresponding physical register. The only difference in the calculation of loop mode synchronization and dynamic mapping-execution timing is the first step. Thus, two similar circuits for the delay analysis have to be used for both timing bits. Also, two delay registers of each physical register are necessary to save the pico-cycle delay times.

Another timing issue to be considered is the interconnection delay that reaches its maximum when the branch unit reads the result of the rightmost FU. The same occurs also when the LD/ST unit reads the result of leftmost FU in the array. In contrast, the delay is at its minimum when a FU reads the result of the previous FU on the same column. This delay does not feature an obstacle in terms of electronic input hazards resulting from the different arriving times of both sources, since the execution is asynchronous. However, it is necessary to set a timing scheme that indicates at which time the result is available. To scale with this latencies, an array width reduction is introduced in Section 4.3.1. However, the maximum delay on the interconnection is considered in the frontend to adjust the timing scheme accordingly. As delaying the token signal more than needed by the data calculation does not affect the correctness of the results at the boundaries of the array.

3.3.3 Branch Handling

Branch Controller

The branch control unit receives the evaluation result of the branch conditions and controls the jump to a new address. The branch controller compares the new address with the address of the first instruction in the array in order to find out if the part of the code mapped to the array is a loop or not. If it is not matching, the new address will be delivered to the fetch unit. If a loop is detected, the execution continues inside the array and all other stages of the processor stalls. After the execution of the loop is finished, the execution continues outside the loop until the fetch unit receives an outstanding address from the branch controller (due to a taken branch or misprediction in the case of implementing branch prediction). Another scenario is that the execution continues until the array is full and the configuration unit can not map any new instruction. In this case the configuration unit signals a full array and waits until the execution of opera-

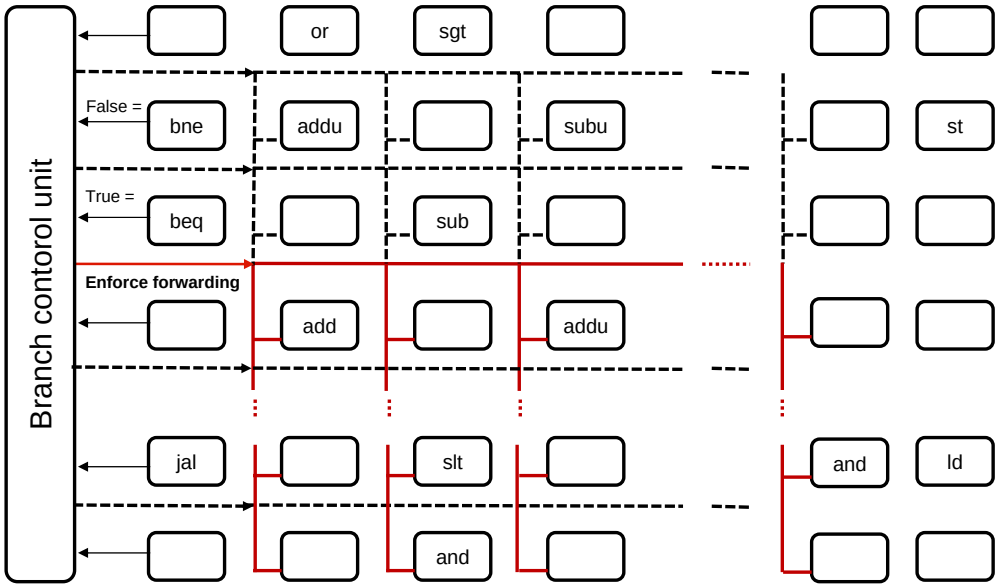


Figure 3.11: The out-of-order execution in the grid

tions is finished. During the write back of results to the top registers (register file) the configuration unit starts mapping new configurations to the array again.

The branch controller is connected to a branch unit in each row of the array, where the branch instructions are evaluated. Additionally, the branch controller is provided by a bus to deliver the new program counter (for taken branches) to the processor frontend together with a *valid*-signal. Besides controlling the branches, this unit enables the top registers to store the values delivered by the feedback network when the execution inside the array finishes. The execution inside the array finishes either when a taken branch is detected or when the array is full and all token signals have been arrived at the end of the array i.e. when all timing tokens arrived at the last row.

With a false estimated branch instruction, the execution proceed in the array without any control reaction from the branch controller. In contrast, by a taken branch, the branch controller receives a signal from the holding branch unit with the jump address. Continuously, the jump address with the validation signal are sent to the fetch unit.

Simultaneously, an *enforce-forwarding* signal is set to cancel all calculated results of the instructions after the taken branch as shown in the Figure 3.11, where all instructions following a branch are mapped to the underlying rows (as mentioned previously). Doing this we kept the out-of-order execution, where all instructions inside the array start their execution when the input values are available. The correct results are then delivered to the top registers by enforce forwarding of all ALUs following the taken branch instruction. The write back to the top registers is also activated by the branch controller. However, the time needed for forwarding the correct results to the end of the array differs based on the row in which the taken branch is executed. Therefore, the branch controller waits until all token signals have been arrived the end of the array and then activates the top registers taking into account the delay of the backward connection wires.

Branch Prediction

The performance of GAP processor benefits especially from accelerating the loops inside the grid. For this purpose branch prediction is an important issue for coarse grained architectures. Branch prediction not only eliminates the control flow inside the loops but also removes the part of the code that is not going to be executed. Control flow elimination allows to capture the loops inside the array when the branch instructions inside the loops are correctly predicted. Without using a branch prediction, a new configuration phase starts each time a branch is estimated to be taken. The configurations of the new configuration phase replace the old ones and prevent capturing the whole loop inside the array.

Loops with control flow contain usually instructions that are not going to be executed depending on the branch instruction. For example, *if, then, else* control change code structure leads to execute one of the blocks, either *then* or *else*. A correct branch prediction of the *if* instruction removes the part of the code that is not going to be executed

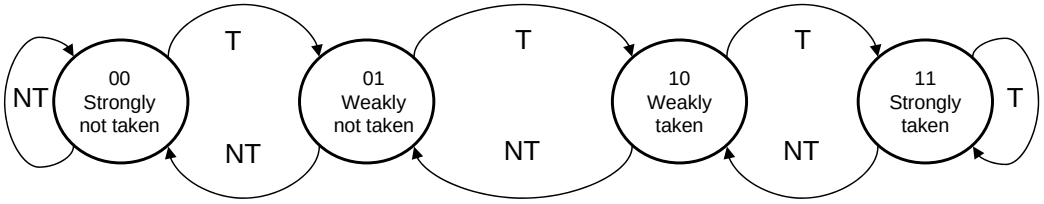


Figure 3.12: state diagram for the branch prediction of direct branches

from the pipeline. This also increases the probability of capturing the loop inside the array, since the array comprises a fixed number of FUs, where big loops can not be captured (especially when both paths have to be mapped like in predicated execution techniques).

Direct branches are treated simply in GAP processor, where the hard to predict branches are handled by the layer optimization presented in Section 4.4. Indirect branches are handled in GAP differently regarding the requirements of the special design of the processor and the mapping mechanism to the array.

Direct branches: like *bne, beq, jump, jal* are handled in the fetch unit with a simple bimodal branch predictor, since the target address—or the distance to the target address—is already encoded in the instruction. The state diagram of the saturation counter with two bits is shown in Figure 3.12. The state diagram shows the transformation steps with each evaluation of a branch instruction. Branches that are always taken evolve the state of the machine toward strongly taken. In contrast branches that always show the behavior of not taken evolve toward weakly not taken state. The advantage of this two bit prediction scheme is that the branch must deviate twice from the last state to change the decision.

The biased branches that are hard to predict are effectively handled by the layer optimization. Thus, a complex branch prediction scheme is not necessary for this architecture. Therefore, no branch history table or pattern history tables (BHT, PHT respectively) are implemented. Less is necessary to implement two level or more complex

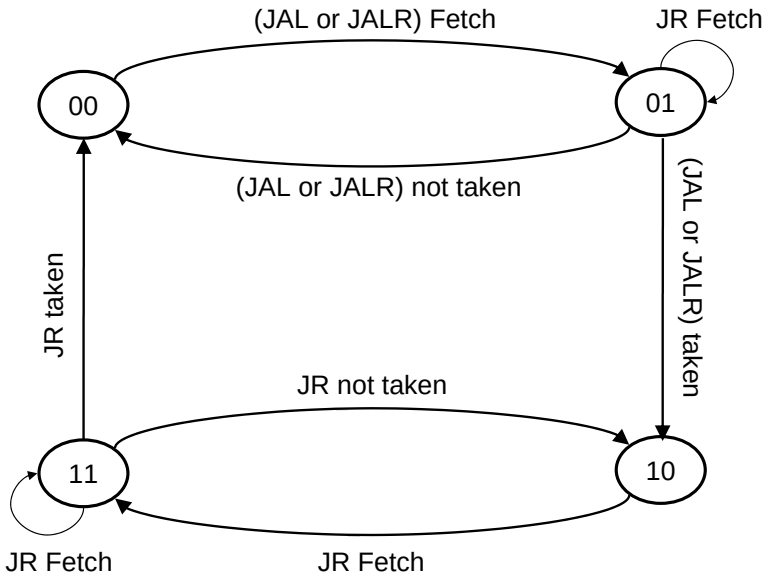


Figure 3.13: state diagram for the branch prediction of JR

neural branch predictors.

Indirect Branches: refers to the target address by means of the number of the containing register, which make difficult to predict it (like jr, jalr). Therefore, we treated functions return jumps i.e. *jr* separately in the fetch unit. A return stack buffer is used to save the addresses of the next instruction after calling a function by *jal or jalr*. A function call instruction in the fetch stage changes the state of the machine as shown in Figure 3.13 to the temporary state $\{01\}$. If the execution finishes in the current configuration phase without to arrive the function call instruction, the state returns to $\{00\}$. This occurs when one of the previous branches to the function call was a misprediction, which changes the flow of the fetched and mapped tile of the program. However, if the configuration phase ends due to a full array (this mean that the function call is still in the pipeline and not mapped yet to the array), the state stays unchanged and waits until the function call is executed in the next configuration phase. If the function call is executed in the array of FUs the state changes to $\{10\}$. A *jr* is allowed to proceed with the prediction only if the latest entry in the return stack buffer is $\{10\}$. The return instruction

in the fetch unit changes the state to $\{11\}$ and to the initial state if the *jr* is taken in the array. However, a non-taken *jr* again according to a misprediction in previous mapped branches changes the state to $\{10\}$.

3.3.4 Data Cache Access Scheme

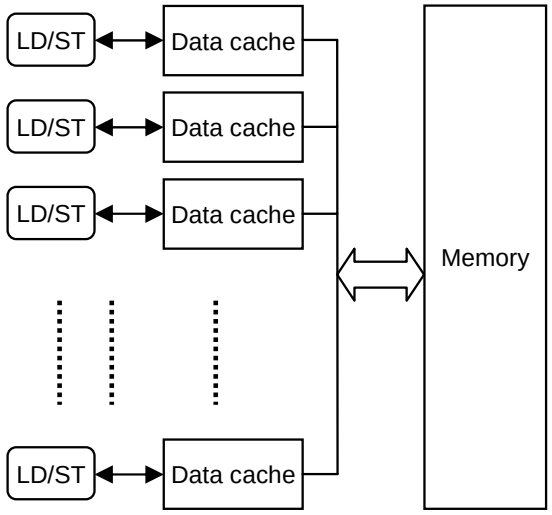


Figure 3.14: Token signal to solve memory access conflicts

Data access parallelization in GAP is done by implementing several load/store units that can access their own relatively-small data caches in parallel as shown in Figure 3.14. Several load/store units are on the first hand necessary to supply the arithmetic/logic operations inside the array with the data from the memory. On the second hand, load/store units together with arithmetic/logic units and branch units allow the mapping of conventional data flow as well as control flow tiles of the program. Each load/store unit can be configured by a memory access instruction. As the execution of the operations is out-of-order, loads and stores can request the data as soon as the execution of the previous operations (on which they depend) have been completed. However, in order to avoid WAW, WAR, and RAW hazards, a store access starts after all previous loads/stores have

been finished and the successive loads/stores must wait until the store finishes writing. This is realized by another token signal that flows through the load/store units only and indicates a possible data access.

Each load/store unit is able to access its private data cache. The data caches are kept different such that maximally one copy of the data exists in one of all data caches accompanied to the load/store units. This organization eases the access to each data cache and removes the duplications of the data sets in the first level cache. Thus, if a store access is a hit in its private cache, then it is not necessary to send the data on the bus to other caches. In the case of a miss in the private data cache, the data is then sent on the bus to update the value that can eventually exist in another data cache. Moreover, by adopting the write through technique, the data must be also sent to the memory. During a store access all following loads wait until finishing the store. Therefore, by a store access, the token signal is hold for one clock cycle in order to ensure reading the correct data by other loads.

Load access completes if the request results in a hit in the accompanied private data cache. However, if a load access incurs a miss in the dedicated cache the request continues on the bus to other D-caches (with short delay in the case of a hit) or to the memory hierarchy in the case of a miss in the first level (first level: all data caches accompanied to the load/store units).

As soon as possible requests accelerate the memory accesses especially inside the loops that are already mapped to the array, where many loads/stores are ready to execute. If these memory access operations are independent of the instructions inside the array, then the requests start immediately. Otherwise, the accesses start as soon as the calculation of the operations (on which the memory accesses are dependent) has been finished, where the execution of the arithmetic/logic instructions is asynchronous. Hence, the asynchronous execution accelerates the execution of the critical path to the memory access to enable starting the request as soon as possible. Moreover, in-order

completion of the memory accesses keep the D-caches consistent with small restrictions on the cache level parallelism in the store case and without the need for a complex data cache coherence protocol. Thus, this data cache organization ensures high parallelism for data intensive applications by servicing several loads in parallel and leads to a better ILP exploitation for basic blocks in data flow as well as control flow dominated applications. A detailed explanation of the memory disambiguation with the memory accesses scheme in GAP is presented in Chapter 5.

3.4 Evaluation

3.4.1 Evaluation Methodology

Parameter	SimpleScalar	GAP
Fetch/Decode width	4-way	4-way
Issue/configuration width	4-way	4-way
Issue/configuration	out-of-order	in-order
Bypass delay/write back	0	1
RUU	64	-
Multipliers	1	1 x rows
Integer ALUs	8	rows x columns

Table 3.2: General parameters of the processor for GAP and SimpleScalar

A cycle accurate simulation environment for the GAP architecture is developed to offer precise simulation results of the examined architecture. To compare with the performance of superscalar architectures, the simulation environment of the GAP architecture is using the same instruction set as the SimpleScalar. The superscalar architecture is chosen as comparison with the GAP architecture, since the GAP processor comprises

Parameter	SimpleScalar	GAP
Branch prediction	bimodal	bimodal
Misprediction penalty	3	3
Return address stack	16	16
Branch target buffer	none	none

Table 3.3: Branch prediction parameters for GAP and SimpleScalar

a superscalar frontend and targets sequential applications. Consequently, a comparison between the simulation results of the GAP-simulator and the out-of-order SimpleScalar is presented on single threaded sequential applications.

The SimpleScalar simulator [56] is used with as similar as possible configurations but less functional units for comparison, as increasing the number of functional units above eight in the SimpleScalar does not show any acceleration. Moreover, the implementation complexity of an out-of-order processor with more than 8-way issue/execution stage makes the design unreasonably complex. The configurations of both simulators are listed in Tables 3.2, 3.3,3.4.

The bypass delay in the SimpleScalar is compared to the write-back of the results in GAP, where writing the results to the top registers in the array takes one clock cycle. The SimpleScalar is simulated with one multiplier, where one multiplier in each row of the GAP-array is assumed for the simulation. The optimization of the number of multipliers and the hardware specification is discussed in Section 4.2.

In Table 3.4, the data cache size of both simulators is the same size. The GAP is simulated with a 1 KB data cache accompanied to each load/store unit (each row). Thus, for example, 4 row-array yields simulating the GAP with a 4 KB data cache and is compared to SimpleScalar with 4 KB data cache. To that, the design reasonability of the GAP does not imply the deployment of a load queue as in out-of-order processors. The characteristics of the data caches in GAP is discussed in details in Chapter 5.

Parameter	SimpleScalar	GAP
L1 I-Cache	128:64:1	128:64:1
L1 D-Cache	1KB x GAP-Rows	1KB for each L/S unit
L1 cache ports	2	1
L2 cache	-	-
TLB	-	-
Data cache queue	LSQ 128 entries	SQ 128 entries
Cache latency	2	2
Memory bus width	16 bytes	16 bytes
Memory ports	2	2
Memory latency	24	24
replacement strategy	LRU	LRU

Table 3.4: Memory parameters for GAP and SimpleScalar

As testing benchmarks, the MiBench benchmarks [57] are selected to offer a performance comparison on different kinds of sequential workloads. The MiBench contain several workloads from different application fields like: automotive, consumer, network, office, security, and telecommunication. Hence, a comparison between the performance results gained from simulating selected MiBench benchmarks on the GAP-simulator and on the out-of-order SimpleScalar is presented in the next section.

3.4.2 Performance Evaluation on GAP-simulator and SimpleScalar

The simulation results of GAP-simulator and SimpleScalar are shown in Figure 3.15, where GAP is simulated with 32 rows and 32 columns. It can be recognized that highly sequentialized benchmarks like *sha*, *rijndael-decode*, *rijndael-encode*, and *jpeg* perform

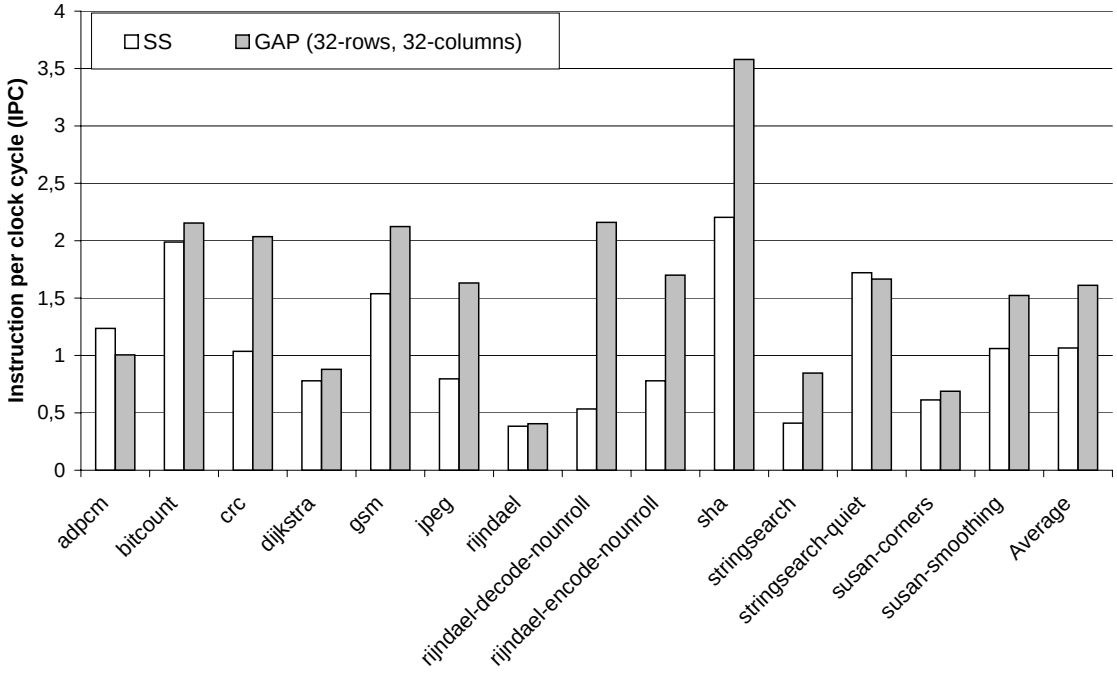


Figure 3.15: Instruction per clock cycle (IPC) for SimpleScalar and GAP (32 rows and 32 columns array) on MiBench benchmarks

much better on the GAP simulator. On average the GAP-simulator achieves a speed up of 1.5 comparing to SimpleScalar on the tested benchmarks. The *adpcm* benchmark shows a reduced performance on GAP-simulator according to hard to predict branches inside the loops. A misprediction inside a loop results in a new configuration phase and replaces the previous parts of the loop that are already mapped. This prevents the capturing of the whole loop inside the array and withdraws the most important capability of the GAP, namely accelerating the loop execution. In Section 4.4, the extension of the GAP architecture is presented to work around the hard to predict branches to enable capturing the whole loop inside the grid even when a misprediction occurs.

All benchmarks comprising one or more loops benefit from the loop acceleration in the GAP array. The array exploits high ILP and memory access parallelism by execut-

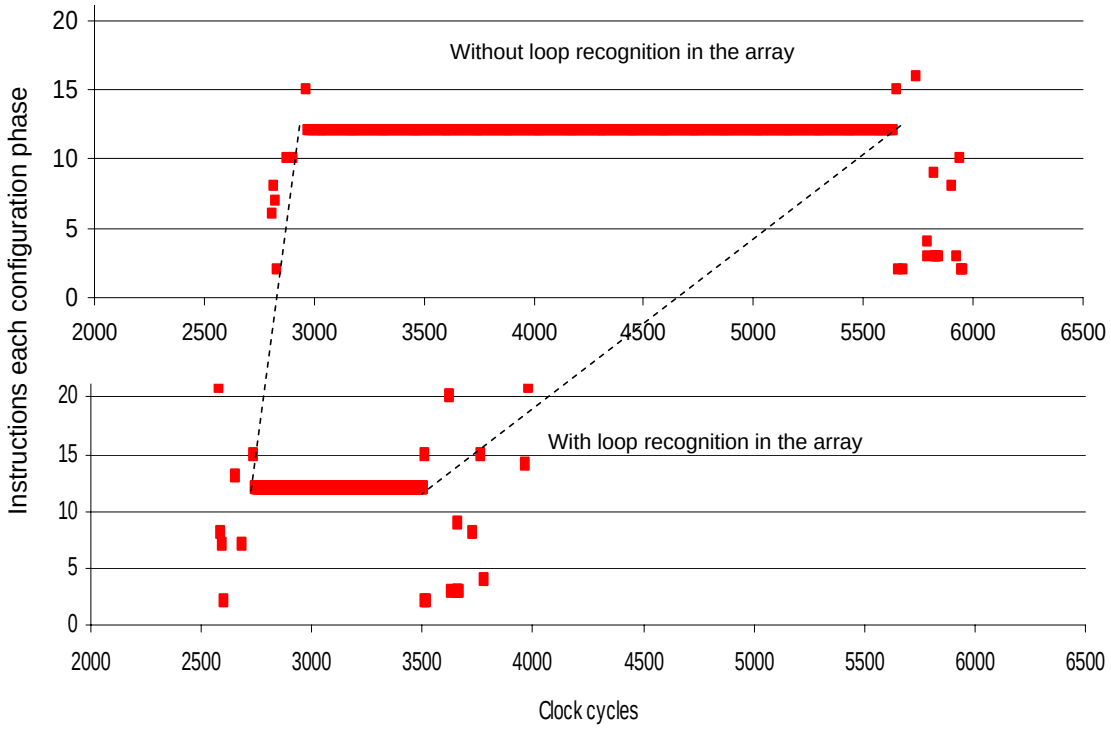


Figure 3.16: Loop acceleration in GAP

ing the loops inside the grid. Moreover, it achieves a speed up by executing the loop body asynchronously. Figure 3.16 shows an example for the execution time of a loop taken from *sha* benchmark on GAP. The figure represents the needed clock cycles for the loop execution without to save the configurations in the array on the upper part of the figure (the loop body is mapped again to the array in each iteration), whereas, on the lower part of the figure, the configurations are kept inside the array and the loop detection is activated. In each clock cycle, maximally four instructions are configured and executed in the first case, whereas, in the second case the whole loop body is saved in the array after the first mapping and the execution continues inside the array for the rest of iterations. The execution of the loop to be recognized on the figure by the steady equal number of the executed instructions each configuration phase. The loop contains 12 instructions, which is relatively small. However, a speed up of 2.3 is achieved.

3.4.3 Evaluation with Different Number of Rows

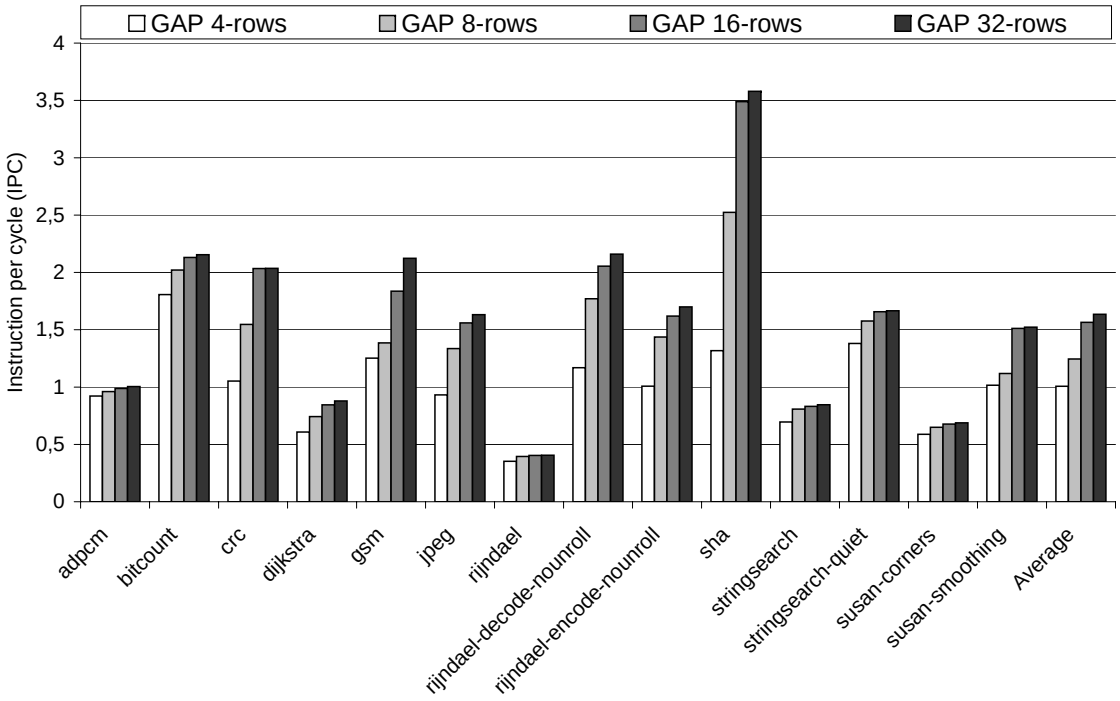


Figure 3.17: GAP performance with different number of rows and 32 columns on MiBench benchmarks

Figure 3.17 shows the performance of GAP with the deployment of different rows in the array. Changing the number of rows does not need more effort than readjusting it in the configuration unit accordingly. The simulation presented in Figure 3.17 shows that the best performance is reached with 32 rows, nevertheless an array with 16 rows achieves a performance very close to the one with 32 rows. The number of rows directly influences the performance of the benchmarks that contain big loops with high grade of data dependency. As big loop need high number of rows to enable mapping the loop body in a single configuration phase to the array. Loops that contain instructions with high data dependency require also an array with high depth to be able to capture the loop, as each instruction with data dependency to a previous mapped instruction must

be mapped to a lower row in the array. The *gsm* benchmark for example achieves 23% more performance with 32 than with 16 rows.

Reducing the depth as well as the width of the array simultaneously is shown in the next chapter. Moreover, many optimizations are also presented to amortize the reduction of the performance resulting from the deployment of small array dimensions.

Chapter 4

Hardware Optimizations

4.1 Introduction

Managing the hardware resources of coarse grained reconfigurable processors in the software prevents undertaking many dynamical reactions needed by the reconfiguration task at runtime to be adaptive with the dynamic program execution. Therefore, we introduced a hardware-based reconfiguration unit with the functionality of mapping instructions according to data dependencies into the array of functional units. The described reconfiguration unit requires some information about the underlying hardware structures. The required information is only the number of rows and in which row the operands of an instruction are available. Using status register to save the row in which an operand is available and update it each time the operand must be read or written makes the mapping task easy. However, the simplicity of the reconfiguration unit moves the burden on the grid to deploy more hardware resources. The presented mapping strategy leads to a correct execution only when the number of columns in the array is equal to the number of physical registers. Moreover, each functional unit must be able to read the result of a functional unit in the last row, i.e. with 32 columns the number of interconnections between two rows is 1024 interconnection each 32 bit. This immense

hardware overhead to keep the interconnectivity is not the only problem that faces grid architectures but also the number of gates to be driven by a single signal (fan-out and fan-in problems). In this chapter, we offer a solution to optimize the hardware resources without to change the data-flow similar execution nature, the simple functional units, and the asynchronous timing inside the grid. The solution is based on improving the reconfiguration functionality to be able to simplify the underlying hardware structures in the array. However, the need for more flexibility to manage the underlying hardware structures increases the demands on the reconfiguration hardware unit. We present also an improvement of the configuration unit for the GAP processor based on the optimization changes that are applied to the grid. The mapping task is improved to be aware of the data dependency of the instructions, complex operation execution, interconnection usability, and the usability of grid regions. All our optimizations in this chapter are implemented to the GAP processor. In general, the optimizations of the grid are also applicable to the grid of coarse grained reconfigurable processors, since the execution inside the grid exhibit similar behavior. However, the special design characteristics must be taken into account during the implementation of these optimizations to other architectures, especially when routing components are attached to the processing elements on the grid.

4.2 Hardware Specialization

4.2.1 A Special Unit for Multiplication/Division

Hardware specialization offers an architectural possibility to reduce the area utilized by the array. Functional resources that take long latencies and large areas can be shared

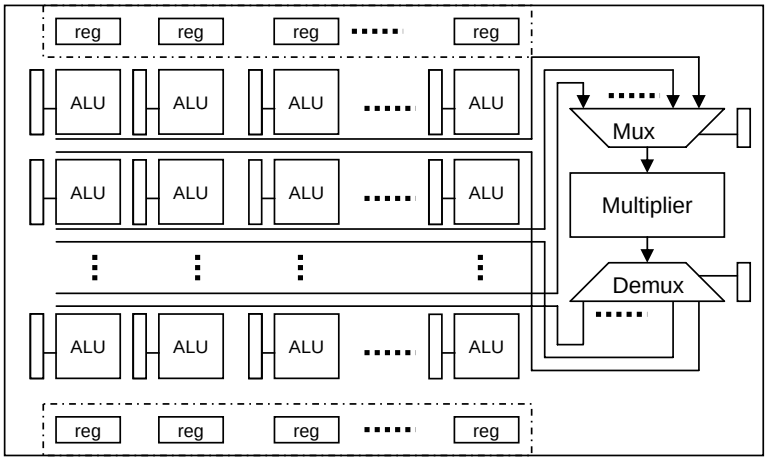


Figure 4.1: A shared multiplier/divider with an array of simplified ALUs

among the FUs on the grid. The basic GAP architecture comprises plenty of functional units with the ability of executing simple integer arithmetic/logic operations and a special column for multiplication/division units. Implementing a single integer multiplier/divider unit instead of a unit in each row highly reduces the hardware cost and has a negligible effect on the performance as shown in the next section. However, this optimization must be taken into account if the software optimization is trying to pipeline the loops with multiplications/division instructions. In this case, loop pipelining can not help mapping different iterations into a single configuration phase. The special multiplication and division unit can read its operands from all rows with the help of a multiplexer. The result can also be redirected to the consuming execution unit by a demultiplexer. The multiplexer and demultiplexer have to be configured by the reconfiguration unit at runtime as shown in the Figure 4.1. The configuration unit can recognize the multiplication/division instructions and map them to the special unit. Another multiplication/division instruction in the same configuration phase leads to signal a full array and start a new configuration phase.

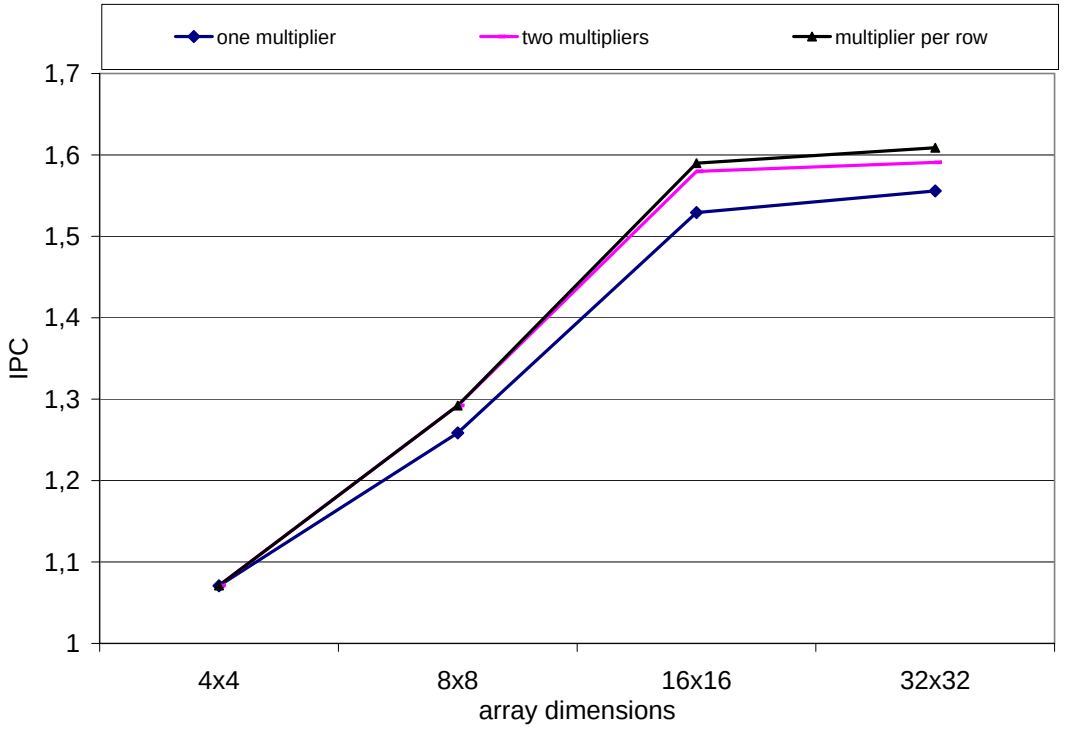


Figure 4.2: The average performance for GAP with different array dimensions and different number of multipliers

4.2.2 Evaluation

The hardware specialization impacts the performance only slightly due to the very small number of multiplications and divisions that can be found inside the loops in the all tested benchmarks. Figure 4.2 shows the average performance of GAP with different array dimensions. The GAP simulated with one and two multipliers and a multiplier in each row as in the basic architecture (we call multiplication/division unit a multiplier). With two multipliers a small difference in the performance can be recognized, the GAP with one multiplier-array can not capture the loops with more than two multiplications. However, the average performance three multipliers aggregates closer to that of an array with a multiplier in each row. The small degradation in the performance is based on the attributes of the MiBench benchmarks, where only some benchmarks are

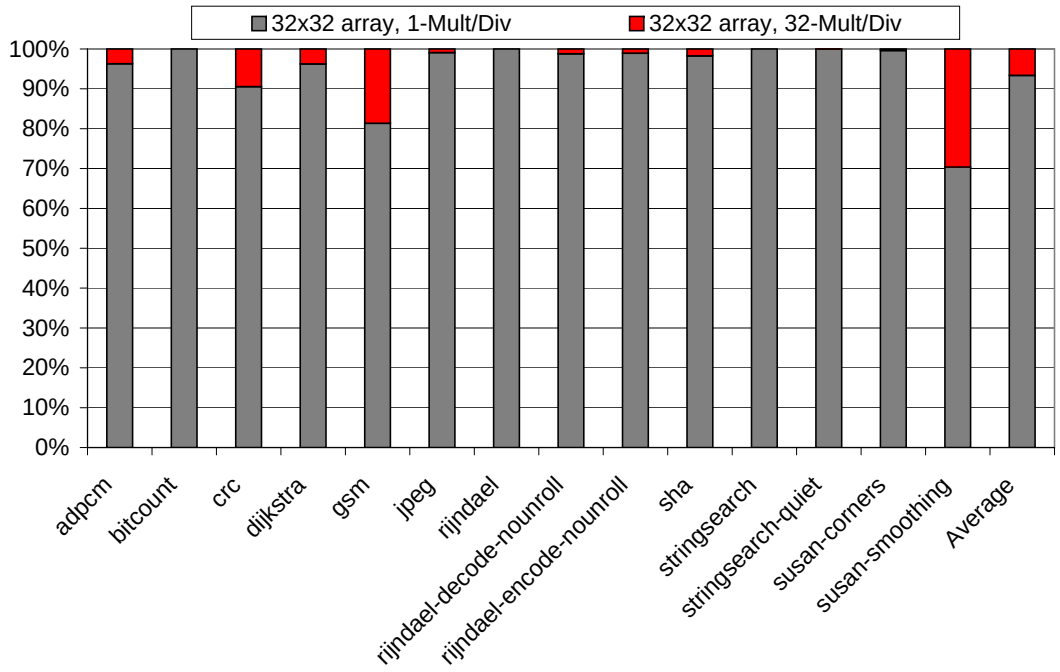


Figure 4.3: The GAP Performance loss with a single multiplication/division unit simulated with a 32x32 array dimensions

executing several multiplication/division operations extensively inside the loops. The performance loss with one multiplication/division unit differs based on the attributes of each benchmark as shown in Figure 4.3.

The performance with small array dimensions is similar for all number of multipliers implemented in the array, since the most of the loops do not fit into the small array even with high number of multipliers. However, increasing the array size emphases the importance of the number of multipliers. Loops that contain more than one multiplication/division operations can be mapped to multiple configuration layers in the array as explained in Section 4.4. Therefore, the performance loss resulting from hardware specialization is marginal. The simulations done in the rest of this work underlay a GAP-array with simple functional units and one integer multiplication/division unit.

4.3 Array Geometries

4.3.1 Columns Optimization Technique

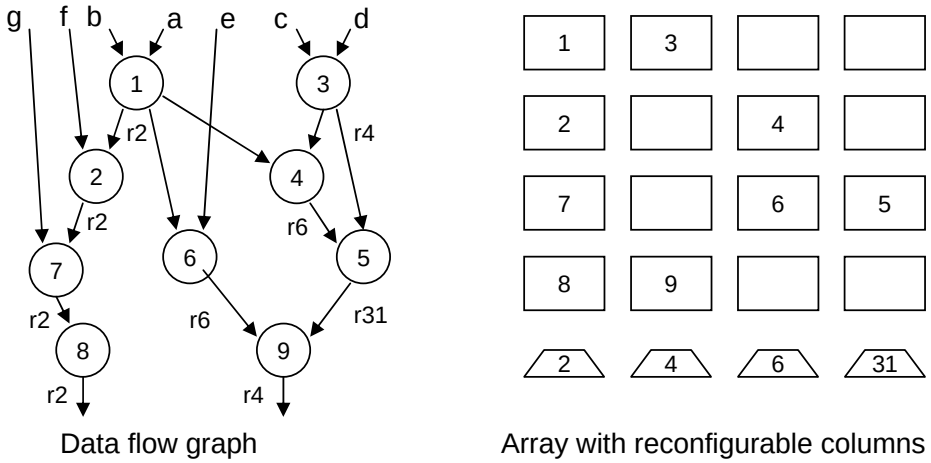


Figure 4.4: The mapping of instructions into a 4x4 array

Each column in the array of GAP processor can execute only instructions that write a single specified destination register, namely the architectural register at the top of the column. This restriction allows many simplifications for the hardware as:

- The reconfiguration unit can easily select the column for the instruction to be mapped by means of the destination register.
- The forwarding of the results on the grid can be simply implemented by means of multiplexers that have to be configured by both sources of the instruction.
- The register file at the top of the array is accessed only twice each configuration phase avoiding the attachment of a register file to each functional unit. Only one reading access to the registers is necessary at the beginning of a configuration phase and a one writing access when the execution completes.

Assigning each column in the array to a specific physical register in the presented GAP architecture features an ineffective hardware implementation. This leads to a number of columns in the array equal to the number of architectural registers—typically 32—and increases the hardware costs. In this section a solution for the reduction of the number of columns is presented. The main challenge of this optimization is to keep the data-flow-execution nature inside the grid, the asynchronous execution constraints, and the simple result deliverability between the FUs.

To reduce the number of columns we have improved the mapping mechanism in the reconfiguration unit to be able to reconfigure the result write-back of each column in the array. For this purpose, we enhanced each column by a reconfigurable demultiplexer to redirect the result to the matching register (see Figure 4.4). Thus, if a column is reconfigured to a destination register x, only instructions that write x will be mapped to this column during the same configuration phase. If an instruction with new destination register—that is still not assigned to a column—has to be mapped, the reconfiguration unit reconfigures a free column to the specified register. In the case where no column to be reconfigured is available, a full array is signaled in the configuration unit to delete the current configurations and to start the mapping of a new configuration phase. In this way, we have kept the simple array consisting only of reconfigurable functional units and interconnections without to attach a router or register file to each FU.

Figure 4.4 shows the mapping of small data flow graph—on the left side of the figure—to an array of 4x4 FUs and the accompanied reconfigurable demultiplexers. The destination register of each operation is shown on the data flow graph. Demultiplexer reconfiguration takes place in parallel to the mapping of the first instruction on the same column. In each clock cycle four instructions are mapped to the array. The first instruction shown on the data flow graph is mapped to the first column and the accompanied reconfigurable demultiplexer is reconfigured to redirect the result to the same destination register of the instruction. Second instruction is mapped to the same column, since it writes the same register of first instruction. Instruction 3 is mapped

to another column and the accompanied demultiplexer is reconfigured accordingly. Instruction 4 writes again another register and has dependency to instruction 3, therefore it is mapped to a new column and a different row. In the next clock cycles the other instructions are mapped following the same way until the array is full—either no rows or no columns are available for mapping new instructions—or a misprediction occurs during the execution. The data-driven-alike synchronization is also kept by this mapping to allow the correct execution and result delivery of the operations.

4.3.2 Evaluation with Different Number of Columns

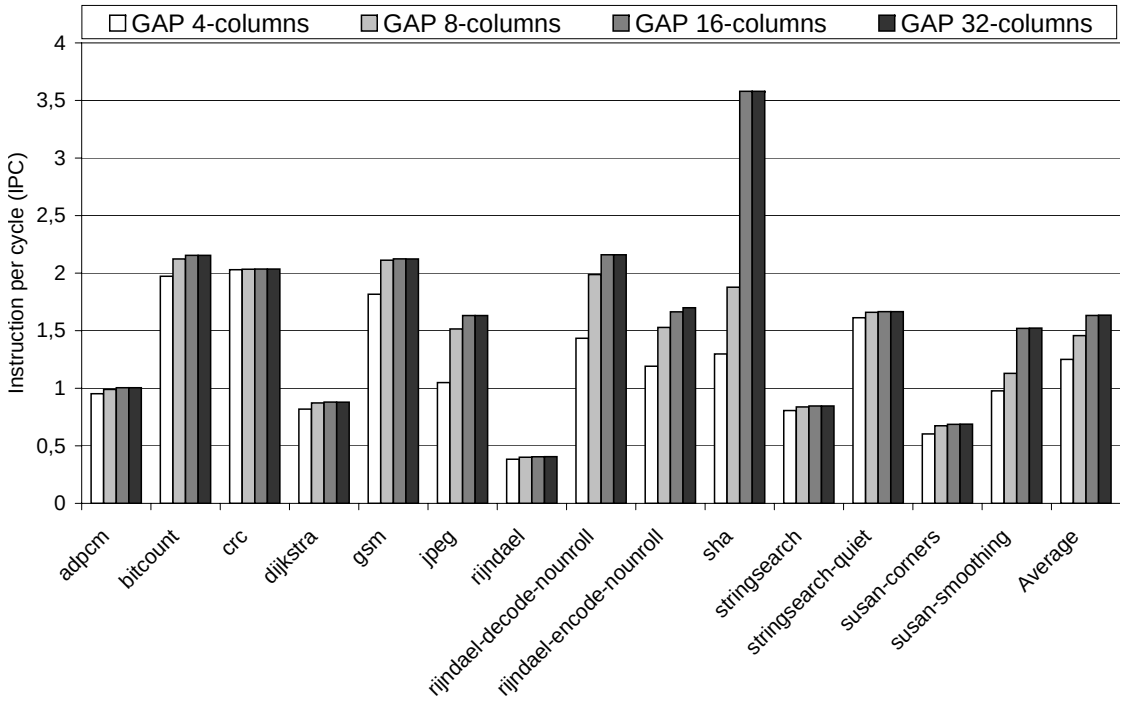


Figure 4.5: GAP performance with 32 rows and different number of columns

Figure 4.5 shows the trade-off between the number of columns and the gained performance. The simulations are done based on an array with 32 rows and different number

of columns. The results show that the highest performance is reached with 16 columns for all simulated benchmarks. Based on these results and the results shown in Section 3.4.3, an array with 16 columns and 16 rows achieves a performance close to the highest performance of GAP. The number of columns plays an important role for benchmarks that contain relatively big loops with instructions that write more destination registers than available columns. This is the case—as could be seen on the figure—for *sha*, *rijndael_decode*, *rijndael_encode* and *susan-smoothing*. These benchmarks show a big improvement in the performance with the increasing number of columns.

Loop acceleration implies the deployment of an array with a number of columns equals to the destination registers written by each loop body. Especially the loop with high ILP can not be captured in the array with smaller number of columns than destination registers of the targeted loop. The benchmarks can not benefit from the high number of rows, if the number of columns is not enough to enable mapping the whole loop body in a one configuration phase. A better overview of the performance with different array dimensions is presented in the next section.

Increasing the number of columns does not show a big effect for some benchmarks like: *adpcm*, *stringsearch*, and *rijndael*. These benchmarks are dominated by several effects that prevent improving the performance. The *adpcm* is dominated by the mispredictions inside the loops, which prevents capturing them inside the array. However, the *rijndael* benchmark comprises several nested loops with function calls that does not fit inside the grid simultaneously. The improving of the performance for these benchmarks is presented with the layer optimization in Section 4.4.

4.3.3 Evaluation with Different Array Dimensions

The column optimization allows us to readjust the simulated number of columns regardless of the number of rows. Thus, an array with variable number of columns and rows

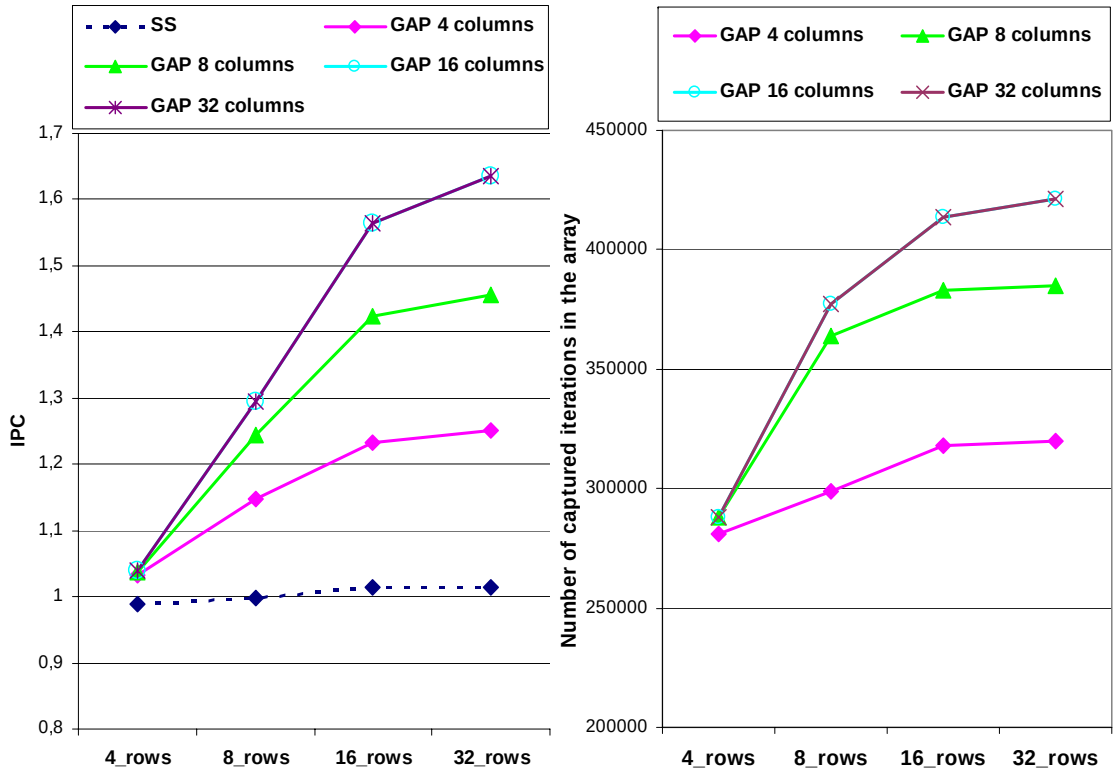


Figure 4.6: Average performance of GAP on MiBench benchmarks, simulated with different array dimensions in comparison to SimpleScalar. The right side represents the number of iterations of all captured loops for each array size

is simulated to gain an overview over the performance with different array dimensions. Figure 4.6 shows the trade-off between the number of columns, rows, and the gained performance in comparison to out-of-order SimpleScalar with different D-cache sizes, since increasing the number of rows in GAP yields the deployment of more D-cache (for both GAP-simulator and SimpleScalar). Increasing the number of columns plays an important role for benchmarks that comprise relatively big loops with high ILP, since loops with instructions that write more destination registers than available columns can not be captured inside the array. The simulation shows that highest performance is reached with 16 columns with all simulated number of rows. Both performance lines

of 16 columns and 32 columns overlap exactly in the figure. This is also approved on the left side of the Figure, where the number of captured iterations of the loops is equal (both lines overlap again).

The depth of the array enhances the performance only if there are enough columns to map the desired instructions and execute them inside the array. The loop characteristics impacts directly the performance, since a highly sequentialized loop features a very poor ILP, but needs less columns on the first hand and it benefits from the asynchronous execution on the other hand, where SimpleScalar can not execute four instructions from the same loop each clock cycle to reach the desired performance. GAP also executes effectively loops with relatively high parallelism; however, the resulting speed-up is then directly affected by the number of the columns.

The number of rows has a direct influence on the performance for benchmarks that contain big loops with high data dependencies. The simulation shows that the best performance is reached with 32 rows. Nevertheless an array with 16 rows achieves a performance very close to the one with 32 rows. Reducing the depth as well as the width of the array can be compensated by deploying configuration layers as shown in the next section.

4.4 Configuration Memories

4.4.1 Related Work

Many research groups have addressed the minimization of the reconfiguration overhead. Much of this work proposes new reconfigurable architectures, like multicontext FPGAs [58]. The designers of coarse grained architectures have also dealt carefully with this issue to accelerate the reconfiguration of the processing elements on the grid.

Multi-context devices allow loading a new configuration while another one is being executed. Afterwards, when the new configuration must start its execution, a context switch occurs that normally can be carried out with small time overhead. This solution drastically reduces the reconfiguration overhead as long as the configurations can be loaded in advance. However, in order to duplicate the number of contexts, the configuration memory resources must be also duplicated, and some additional HW must be added. Hence, the energy of the reconfiguration overhead is not reduced but probably significantly increased. Moreover, a misprediction in the mapped instructions 3 long time delays.

Noguera and Badia [59] have proposed a configuration prefetching approach that attempts to hide the reconfiguration latency on the FPGA. Their proposal is especially interesting because they have developed a HW implementation of a configuration manager that applies their technique providing good results while introducing almost no run-time penalty due to the computations needed to apply it. Other work has proposed a reconfiguration manager specifically designed to hide the reconfiguration latency [60]. This manager applies a prefetch scheduling technique that attempts to load the configurations in advance and a replacement technique that reduces the number of demanded reconfigurations. The manager interacts with a multiprocessor task scheduler in order to obtain accurate information about the near future and use it to take near optimal decisions.

Coarse grained developers usually deploy small configuration memories beside the processing elements to overlap the configuration mapping and the execution [61]. Other architectural approaches implement a configuration caches beside the rows of the processing grid [4], [3]. Also a context memory beside the execution core is implemented in the MorphoSys [5] architecture. The cache is connected to the processing elements with special buses to allow a fast broadcast mapping. Anyhow, the mapping of the configuration to the processing elements incurs an overhead, whereas selecting a cell of the configuration cache on the grid can be much faster. Moreover, the designs that allow

the execution of branch instructions on the grid could incur long time latencies when a misprediction occurs. Moreover, the configurations mapped in advance to the grid must be discarded.

Trace caches [62] are mainly developed for the same purpose of the configuration layers [63]. Traces of instructions—basic blocks or concatenation of multiple basic blocks—are stored in the trace cache to be fetched faster when they have to be executed again. The difference to our configuration layers is that these instructions are stored directly beside the functional unit in our method. Moreover, the trace must be deleted each time a misprediction occurs inside a trace. Differently, we keep the configurations of both paths inside the grid when a misprediction occurs to allow a correct execution of biased branches.

Our configuration memory design envisions a very fast configuration cache beside the FUs. These caches are organized in a disciplinary scheme to save old configurations—in layers—for further use which is the main objective of the instruction caches. In contrast, the introduced configuration caches in coarse grained reconfigurable architecture are used to hold the prefeteched instructions during the execution. A simple layer management policy is introduced to control the switch between the configuration layers.

4.4.2 Configuration Layers

The FU-array features a fixed size that must be area-effective. Therefore, desired code snippets like loops may not fit into the array (as shown in previous section for small array dimensions). To overcome this we developed an array with multiple configuration layers, such that each functional unit in the array can have one or more configuration registers as shown in Figure 4.7. This organization envisions a third array dimension, where the hardware cost of the configuration layers is much less than implementing additional FUs. The implementation of layers is realized as a configuration memory beside

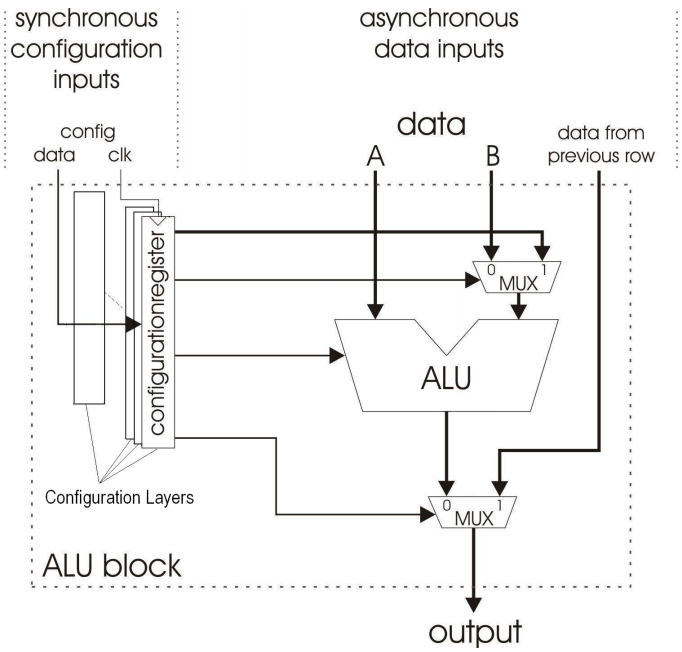


Figure 4.7: Reconfigurable functional unit with multiple configuration layers

each FU and a common signal to select the desired cell as configuration parameters.

This so-called multilayer option targets the reduction of the number of functional units in the array without hampering the performance. It sustains the advantages of mapping whole loops and functions into multiple configuration layers and keeping them for later execution. Many aspects of an effective execution are introduced to the GAP architecture by deploying several configuration layers beside the FUs.

- **Big loops**: With small array dimensions, big loops may not fit into the array. The array becomes full without to be able to capture the whole loop body. After a full-array has been signaled, a new configuration phase starts, where the coming configurations replace the old ones of the same loop. However, implementing several configurations layers with small arrays enable the capturing of the loops even when they do not fit into the array in a single configuration phase. When the configuration unit detects a full array, the mapping continues in another con-

figuration layer without to replace the previous configurations. The execution continues inside the array for the remaining iterations by switching between the holding layers. Hence, the advantage of keeping the loops inside small arrays is kept by deploying enough configuration layers.

- **Nested loops**: Mapping nested loop to a single configuration layer leads to capture only the inner-loop. Each time the outer-loop is executed, the configurations of the outer-loop replace the ones of the inner-loop. The inner-loop must be mapped again with each iteration of the outer-loop, which replaces again the mapped configurations of the outer-loop. With several configuration layers this scenario is avoided, where the inner-loop as well as outer-loop can be automatically detected inside the grid. Finishing the execution of the inner-loop leads to map the coming configurations of the outer-loop in the next layer. Thus, the inner-loop as well as outer-loop are able to be captured in the layers without to replace each other. In the same way, several inner-loops can be handled inside the grid, where the maximum number of inner-loops must be less than the number of implemented layers.

- **Function calls**: Functions can be called from several places in the program during the execution. It is obvious that a single configuration layer is not able to save the function for further executions. However, several layers can enable capturing a function to be executed inside the array during the next calls. It is also not matter whether the function is mapped to one or several layers as with big loops.

- **Mispredictions**: A misprediction inside a loop offers an inconvenience problem to single layer execution. Hard-to-predict branches (biased branches) inside a loop withdraw the most important advantage of the GAP, namely executing the loop inside the array. By a misprediction, a new configuration phase starts and replace the previous mapped instructions. However, with the layer option, the new configuration phase starts the mapping to a new layer. In the next iterations, both

paths after the biased branch are saved in the array. If the prediction of the branch is true the execution continues in the same layer. Otherwise, the execution continues in the next layer, where the second path is already mapped. Thus, saving both paths in the layers eliminates the misprediction penalty. Unfortunately, the switch between layers consumes one clock cycle for writing the result back to the top of the array and changing the control to the desired layer, where a misprediction in a contemporary out-of-order superscalar processor incurs a penalty of more than 10 clock cycles.

- **Multiplication/Division**: Loops that comprise more than one multiplication/division instruction can not be captured in a single configuration phase based on a one multiplication/division unit in the grid. With several configuration layers, the mapping continues in next layer when the multiplication/division unit is busy. This absorbs the negative effect of mapping loops to an array with one multiplier/divider.

The execution in a layer (a configuration phase) finishes when the array is full or a misprediction occurs. The allocation of the layers in both cases follows a simple strategy that always compares the following instruction address—the target address of the branch or the next address to be mapped when the array is found to be full—with the address of the first instruction in each layer as listed in Algorithm 1. If there is a match the control switches to the corresponding layer and continues the execution ($activate(LAY_i)$), whereas the processor frontend stalls. Otherwise, the next layer will be utilized for the new configurations ($allocate(LAY_{curr+1})$). If the current layer is the last one the control moves to the first layer ($allocate(LAY_0)$). Thus, the orchestration strategy for the configuration layers is using FIFO as replacement strategy. The hardware implementation of the FIFO strategy is simple in comparison to other strategies like least recently used (LRU) and delivers very similar results for the tested benchmarks. Other complex algorithms based on compiler analysis—for extracting the gain factor of keeping a function or a loop inside the grid—require the software interaction and are not considered in this work. Replacing one of the layers where a part of a function or loop is mapped does

not feature a difficulty for the execution, since the control of the layers compares the first address of the layer with the jump address before performing a switch to the layer to ensure a correct execution. Thus, even with replacing a part of a function or loop does not prevent from detecting the parts that are still kept inside other layers. Then, the replaced parts must be fetched and configured again in order to complete the execution.

Multiple layers of configurations envision an effective instruction cache avoiding cache line conflicts, since blocks are being replaced in the array instead of instructions in the cache. Deploying multiple configuration layers draws the computation intensive parts of the code closer to the execution units avoiding I-cache accesses and instruction processing in the frontend of the processor.

Algorithm 1 The control switch policy for the execution with layers

if $new\_address = true$ **then**
  for $i = 0$ to N_LAY **do** $\{; N\_LAY$ the number of layers$\}$
    if $new\_address = LAY\_ADR_i$ **then**
      $activate(LAY_i)$
      $stall_{frontend} \leftarrow 1$
      return true
    end if
  end for
  $Fetch \leftarrow new\_address$
  $activate_{frontend} \leftarrow 1$
  if $LAY_{curr} \leq N\_LAY$ **then**
    $allocate(LAY_{curr+1})$
  else
    $allocate(LAY_0)$
  end if
end if

4.4.3 Evaluation

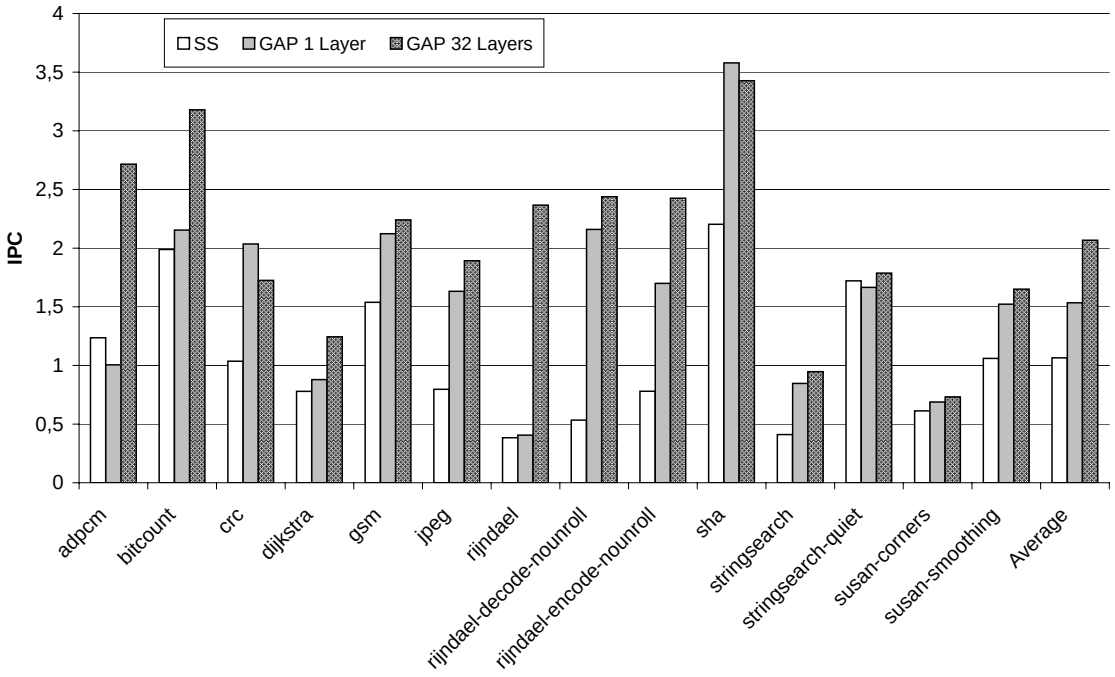


Figure 4.8: IPC for SimpleScalar and GAP with one layer and 32 layers and a 16x32 FU-array

Multiple configuration layers reduce the penalty of a misprediction inside the loops and the functions significantly, since a misprediction switches the control to another configuration layer without replacing the current configurations. However, this solution is not ideal since one clock cycle is needed to change the control. Also, it reduces the number of instructions in each configuration layer, i.e. it reduces the ILP. Nevertheless it offers a very effective solution to avoid the misprediction for those hard to predict branches. Taking a look at *adpcm* benchmark reveals the high number of mispredictions inside the loops that occurs during the execution. As shown in Figure 4.8, the mispredictions of *adpcm* are effectively avoided with the layer-option by keeping the advantage of executing the loops inside the array. Differently, *bitcount* benchmark ben-

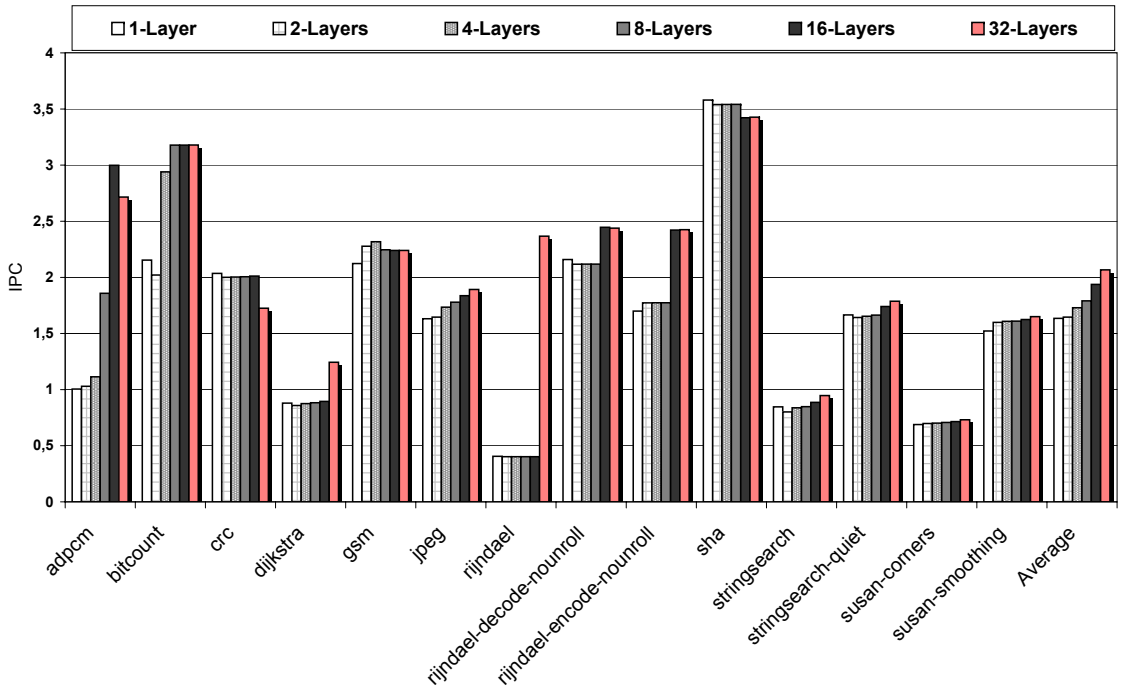


Figure 4.9: IPC for GAP with several configuration layers (1 to 32) and a 16x32 FU-array

efits mainly from saving the functions—which are called many times and from different places in the program—in the layers and executing them repeatedly. The GAP simulator with 32 layers shows a much better speed-up with about 5-fold for *rijndael* benchmark. *Rijndael* is discussed in details in Section 4.4.4. On average, the GAP with 32 layers and a 16x32 array achieves a speed-up of 2.1 over the SimpleScalar.

Figure 4.9 shows the trade-off between the number of layers and the performance. Surprisingly, for some benchmarks a small degradation in the performance occurs with a higher number of layers, e. g. for *adpcm* and *sha* (16 vs. 32 layers). This is incurred by the control flow in the loops and functions. For example, loops with control flow changes are mapped at first time to multiple layers, since a misprediction occurs at first time execution. With enough layers, the loop will be captured and the execution

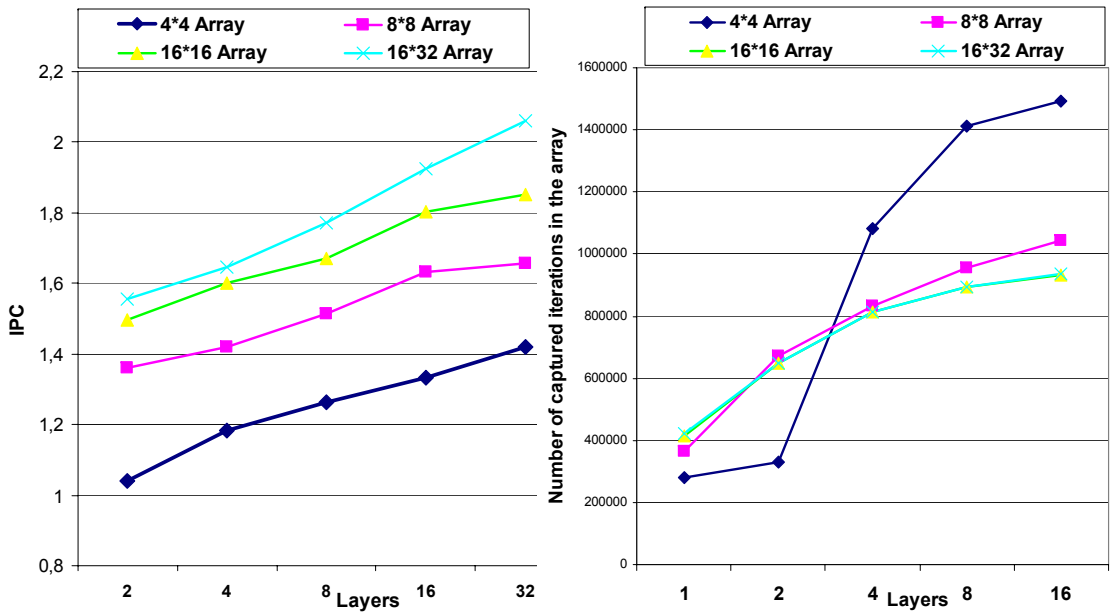


Figure 4.10: Average performance of GAP with different array sizes related to the number of layers. The left side presents the number of captured code blocks in the array for both loops and functions related to the number of layers

continues without remapping the already mapped configurations. These configurations are relatively short and, therefore, have a low degree of ILP. Additionally, penalties occur whenever a switch between two layers has to be performed, where the number of switches is crucial in this case. However, with a smaller number of layers the loop scattered on several layers will at some point be replaced and mapped again. The resulting configurations in each layer might be longer, because the already warmed-up branch predictor predicts the branches correctly. These longer configurations execute faster than small scattered ones, since they exhibit a higher degree of parallelism and spar many control switches between the layers.

Figure 4.10 shows the average performance for MiBench workloads and different array sizes. The performance improvement with 32 layers is about 50% for all simulated

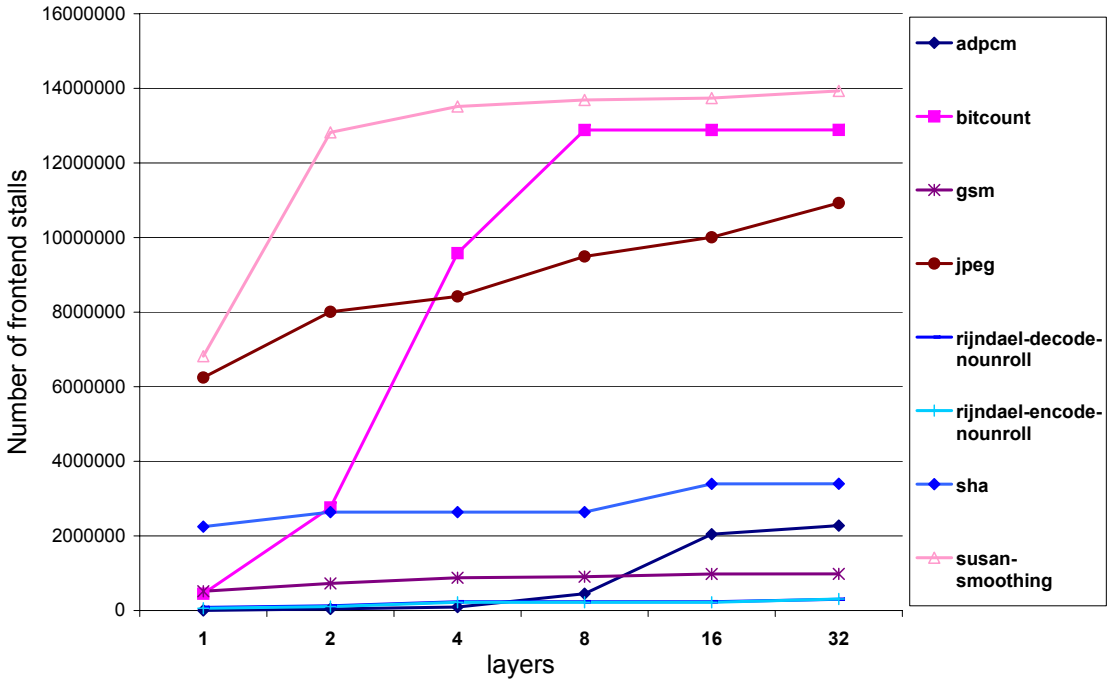


Figure 4.11: The number of stalls in GAP's frontend related to the number of simulated layers

array dimensions. The GAP-simulator with a 4x4 array outperforms the SimpleScalar by about 50%. A 16x32 basic array as well increases gracefully the performance with more layers. However, each benchmark comprises a finite number of loops and functions and hence, implementing more layers than the repeated structures can not speed-up the execution any more. Moreover, the implementing of more layers could mount problems of hardware limitations and layer selection time.

The number of captured blocks of loops as well as of functions with different array dimensions is shown on the right side of the Figure 4.10. The number of captured blocks in a 4x4 array is increasing dramatically with the more number of layers. However, the acceleration does not take the same sharp course of improvement, since the captured blocks are small due to the small array dimensions. However, a small increasing in

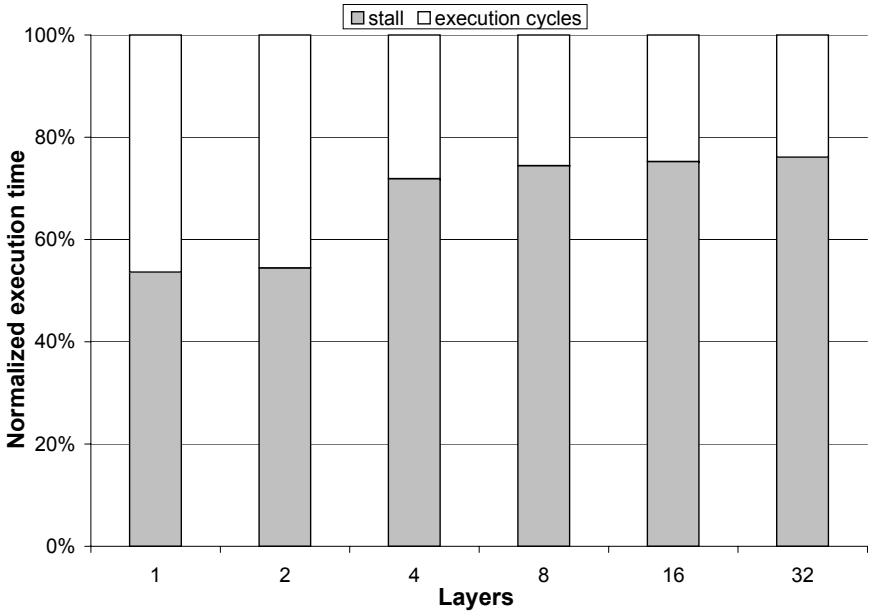


Figure 4.12: Normalized stall time in GAP front-end with different number of layers

the number of captured blocks in a 16x32 array enhances the performance significantly. This is also reasonable based on the potentially high number of instructions in the captured block.

Figures 4.11 and 4.12 show the number of stalls in GAP front-end resulting from executing loops and functions in the layers. The increasing of the stalls in the frontend indicates that more loops and functions are detected in the grid and therefore the frontend is set to idle state. Figure 4.11 shows a great increase in the stall time for some benchmarks with the more number of layers. This is clear for benchmarks with long time execution more than small benchmarks like *rijndael* on this figure. Figure 4.12 shows the average stalling time in the frontend regarding the normalized execution time of all tested benchmarks. Increasing the number of layers with a 16x32 array increases the stall time in the processor front-end for more than 70% of the execution time. The other 30% of the execution time contains also waiting slots based on instruction cache misses and hence, the processor does not pose a challenge in the front-end including the

configuration unit. Moreover, during the stall time only FUs that hold configurations execute and receive power, whilst others are set to idle state.

4.4.4 Discussion of Rijndael Results

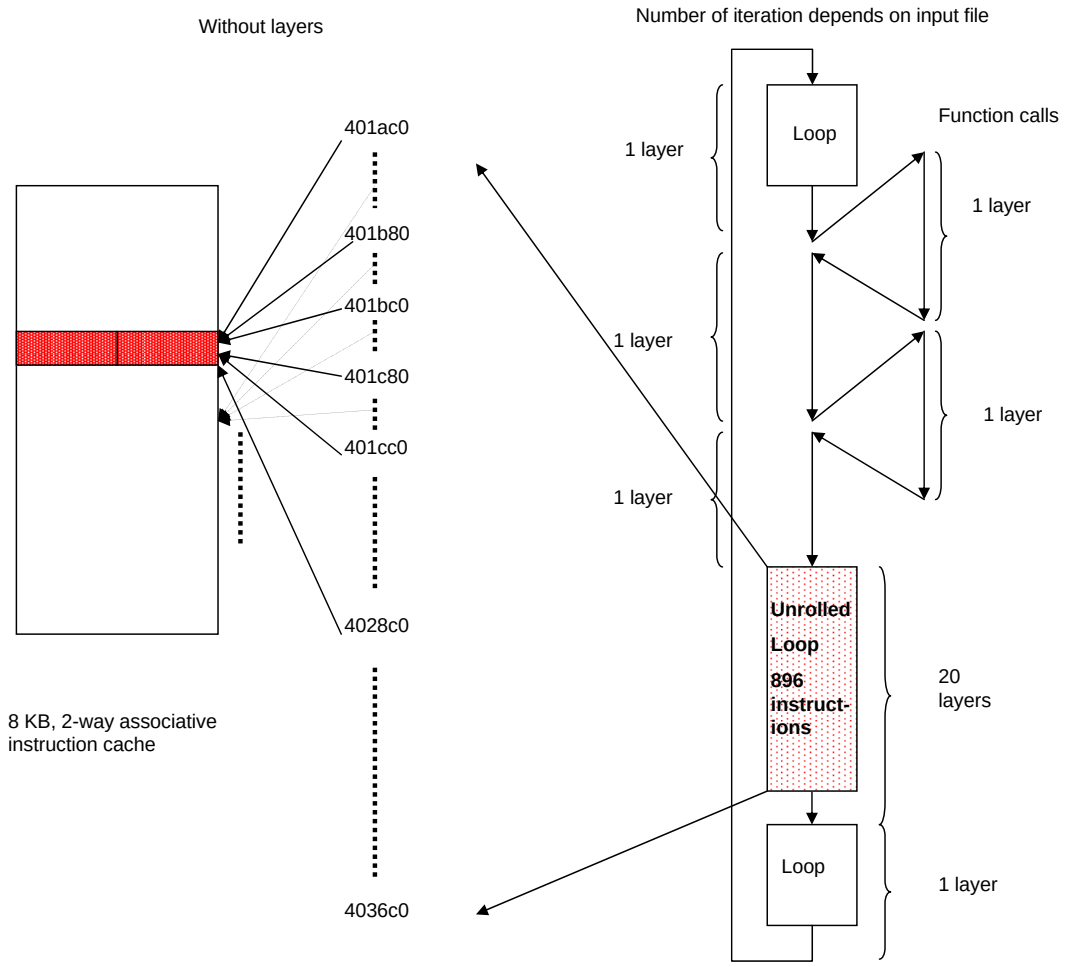


Figure 4.13: Mapping of the computation intensive part of rijndael to layers

Layers exhibit a very important characteristic for avoiding conflict misses incurred by instructions that write the same set in the instruction cache. In contrast to instruction cache, a complete block (configuration phase) must be replaced in the layers when

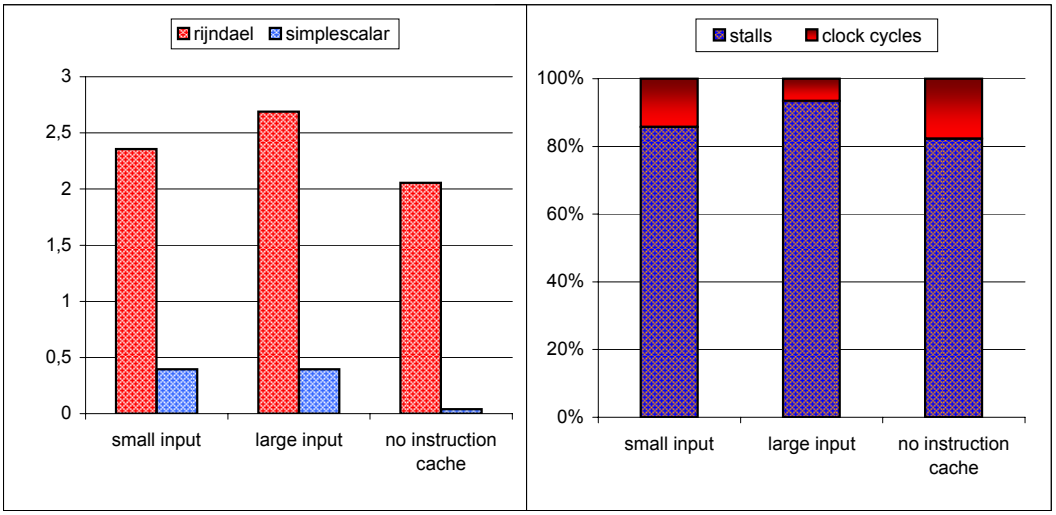


Figure 4.14: IPC and front-end stalls of rijndael using 32 layers

deciding to start a new configuration phase. Thus, unlike the set-based replacement strategy in the caches, the whole targeted block of instructions will be found in one of the layers or not.

Multiple layers of configurations exhibit the behavior of an effective instruction cache beside the FUs. Using multiple configuration layers draws the computation intensive parts of the code closer to the execution units avoiding I-cache accesses and instruction processing in the front-end of the processor. It even helps avoiding time-consuming I-cache misses caused by the permanent replacement of adjacent instructions in the same set as shown in Figure 4.13, where the number of instruction cache misses does not differ in both SimpleScalar and GAP-simulator with one layer.

As a special case study we consider the *rijndael* benchmark with the unroll option, as it shows a surprising speed-up and a reduction of the number of cache misses in comparison to SimpleScalar. Figure 4.13 presents a simplified program flow for the most intensive computation part of *rijndael* and the effect of an unrolled loop on the cache misses. On the right side of the figure is the loop that must be repeated a number of times depending on the size of the input file that have to be encoded/decoded by the

rijndael algorithm. This loop comprises many code structures like functions, sequential segments and other nested loops. The huge number of instructions in the block of the unrolled loop has an essential effect on the number of cache misses—as shown on the left side of the same figure—, where the many instructions inside the block are writing the same set in the instruction cache. Many cache misses are incurred by the permanent replacement of the same cache area by this giant basic block. A GAP array with 32 layers, 32 rows, and 13 columns can keep the configuration of the whole mentioned structures in the layers and executes them as often as the envelope loop must be repeated. Thereby, the GAP avoids a lot of I-cache accesses and I-cache misses. Moreover, the array exploits a high ILP for the captured blocks and executes them asynchronously. It also extracts more parallelism by accessing the data cache in parallel, since a lot of loads/stores and other arithmetic/logic instructions have been already mapped to the array. Thereby, the IPC increases immensely as shown in Figure 4.14.

The number of repetitions of the envelope loop—the size of the input file to be decoded/encoded—increases the execution time inside the grid and result in a better performance than SimpleScalar. Moreover, the instruction cache access in GAP is at its minimum, and hence, the simulation of GAP and SimpleScalar without instruction cache shows a GAP's speed-up with about 50-fold comparing to SimpleScalar. The front-end of the processor stays stalled for about 90% of the execution time as shown on the right side of the figure. This leads to very low energy consumption per instruction, because only the FUs that hold instructions to be executed are receiving power. A further reduction of the array size and the number of layers inside the grid preserving the superior performance of GAP is presented in Section 4.6.2.

4.5 Interconnections and Meshes

4.5.1 Interconnections in the Coarse Grained Reconfigurable Architectures

Several interconnection topologies can be considered during the design of the grid in coarse grained architectures. ADRES processor is presented with several interconnection meshes of the processing elements on the grid. An array exploration with several hardware structural variations is presented in [64] [6]. The different kind of meshes examined does not only affect the performance but also the power dissipation and the register file distribution on the grid. The MorphoSys processor [5] [42] [45] connects the reconfigurable components on the grid by the means of two layers of interconnection networks. The first layer connects the components with a 2-D mesh, whereas the second layer hardwires the quadrant level to provide a complete row and column connectivity within the quadrant. For the data transportation a dedicated 128-bit bus is linked to the column elements on the array. A context bus is also deployed to distribute the required instructions to the processing elements.

The TRIPS processor [3] [65] [66] is designed with an interconnection network that connects the processing elements by the means of routers. The compiler with the analysis and placement tools takes the distance between the data dependence instructions into account to place them as near as possible to each other. Doing this reduces the communication delay on the interconnection network and set up the necessary routing configurations. The Raw processor [4] [67] is based on processing elements able to route the results by four 32-bit full-duplex on-chip networks. Over 12,500 wires are used to ensure the static and dynamic routing. The routers in static mode are specified at compile time, whereas dynamic routing is specified at runtime.

The dense connectivity of the interconnection networks makes it difficult for auto-

matic routing tools to maintain regularity of the global routing. The tools are not only responsible for the data dependence analysis and scheduling based on the underlying interconnection mesh but also the timing and routing configurations. This makes the software configuration performance inefficient and hampers the performance of the whole design. In contrast, our grid design implements very simple FUs on the grid, where the interconnection demands are very high. In the next section, we offer a hardware-aware solution methodology for coarse grained architectures with data-flow-alike execution core to simplify the interconnection network.

4.5.2 Reducing the Number of Interconnections

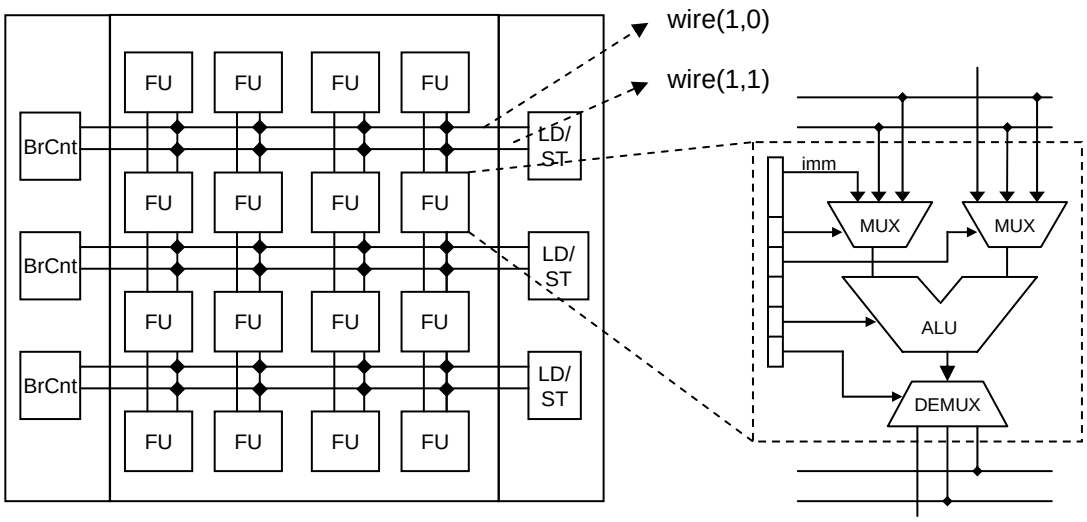


Figure 4.15: A 4x4 example array with two interconnections between the rows and the reconfigurable functional unit with simple multiplexers for input selection and a demultiplexer for output redirection

Designing the interconnection network of the array, the main targets were simplicity, meeting the asynchronous constraints and avoiding direct accesses to the register file. This has been achieved by assigning each physical register to a specific column in the

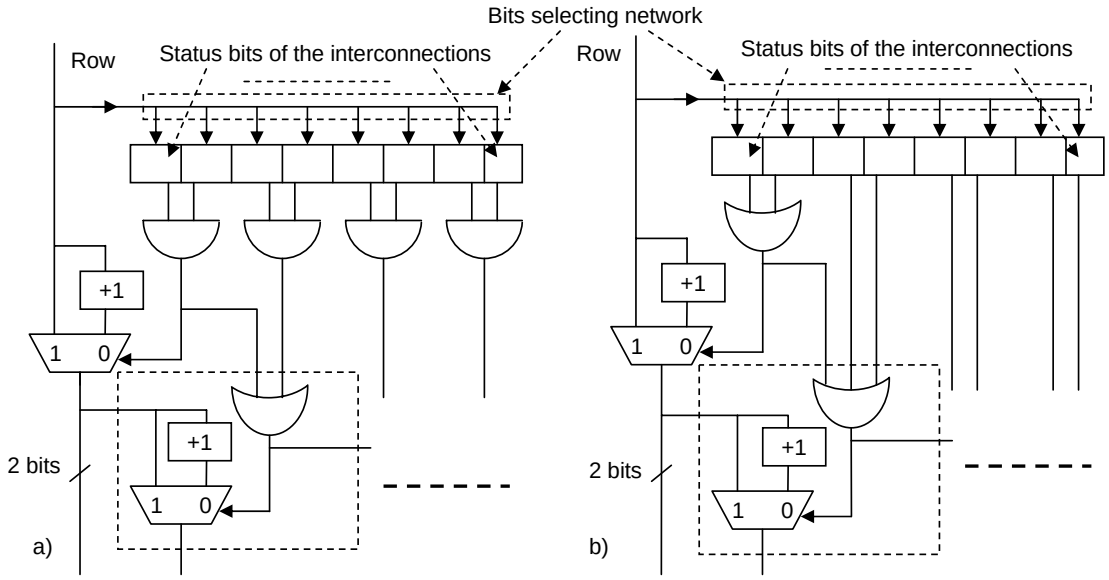


Figure 4.16: Row selection decision based on the status of the interconnections when a) the instruction requires both interconnections, b) the instruction requires only one interconnection.

grid and allowing the FUs to read only the results of the previous row. However, the need for more flexibility to manage the underlying hardware structures increases the demands on the reconfiguration hardware unit.

Extending the functionality of the reconfiguration unit to be able to recognize the busy hardware structures enables also reducing the number of interconnections. The interconnections are marked as busy or free in the configuration stage using status register. When an interconnection is reserved to transport a result between two FUs, it remains busy during the whole configuration phase, i.e. until the array is full or a misprediction is detected, which leads to delete the current configurations. The interconnection is set to be busy during the whole configuration phase as the execution inside the array is asynchronous. Figure 4.15 shows an array with only two interconnections between two rows. The right side of the same figure depicts a FU including an ALU with two

multiplexer—each with three inputs—and one demultiplexer to write the result on the desired output wire. Thus, reducing the number of interconnections also decreases the number of bits needed for the control of the multiplexers and demultiplexers in the grid. Executing an instruction in a FU that adds a register value to an immediate value and put the result in the same register does not require any configurations for the multiplexers (zero for selecting immediate value and zero for selecting the same register value from previous row). However, an instruction that requires a value from a different column as an operand sets a free interconnection to be busy. The reconfiguration unit marks the interconnection as busy during the mapping of the instruction and reconfigures the controlling register in the FU accordingly to read from the specified wire.

Each interconnection is managed in the configuration unit by a status bit as shown in Figure 4.16. The interconnection availability check starts after making the row decision based on analyzing the data dependencies of both operands of each instruction. The left-side circuit of the figure demonstrates the row selection based on interconnection availability if both interconnections are required by an instruction. The right side of the figure shows the decision circuit in the case where only one interconnection is needed. The bit selecting network activates the status bits of all interconnections of the rows beneath the row of dependence decision. The interconnection check circuitry traverses the status bits of the interconnections to find out the next possible row with enough free interconnections for mapping the instruction.

Both circuits shown on Figure 4.16 can be combined in one circuit to enable reading from the same status register. Moreover, the circuit can be implemented in the same stage of the configuration unit or in a separated stage, since the functionality of the circuit depends only on the selected row based on the dependence analysis. Another design possibility can also be considered by implementing the interconnection management circuit in the configuration stage and making it parallel to the dependence analysis. To do that, the dependence analysis and interconnection check for all rows start simultaneously. Afterwards, the next possible row of data dependency decision with enough

free interconnections as needed by the instruction must be selected. However, doing this increases the complexity and possibly affects the critical path delay in the configuration stage. Moreover, the complexity of this circuitry increases with the number of rows in the array, which must also be considered during the register transfer level synthesis.

Reducing the number of interconnections between the rows leads automatically to reduce the fan-out and fan-in of each ALU, the size of the multiplexers, and the required configuration bits for the input selection. However, a small number of interconnections leads to slip the configuration inside the array downside and requires more rows, whereas the same configuration stays more compact with more interconnections. Hence, the array becomes full with fewer instructions but requires less number of columns. Therefore, keeping the current configuration in the array and reconfigure the remaining columns increase the utilization and the probability of capturing more loops as described in Section 4.6.2.

4.5.3 Evaluation

Reducing the number of interconnections is crucial issue to reduce the hardware costs and to solve the fan-in and fan-out problem. This reduction does not only lead to reduce the hardware costs of the interconnections but also the size of the multiplexers accompanied to each ALU and the number of configuration bits that control these multiplexers. However, this optimization can slightly hamper the performance as shown in Figure 4.17. The figure shows the average performance of MiBench benchmarks with different array dimensions and single layer. The simulation results of two and three interconnections are compared to fully interconnected array between two rows. Fully interconnected means that each ALU can read the results of any ALU from the previous row and forward it—or its own result—to all ALUs in the next row. The simulation shows a very small degradation in the performance with two interconnections and non remarkable reduction with three interconnections in comparison to fully interconnected

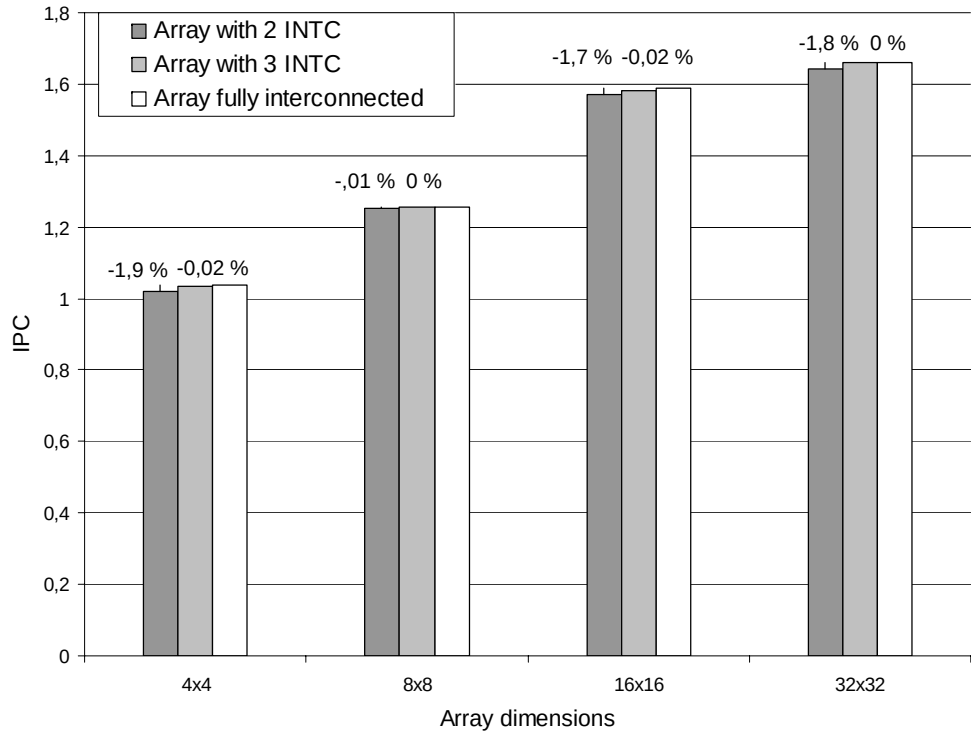


Figure 4.17: GAP average performance with two and three interconnections in comparison to fully interconnected array. The simulation is done with single layer and different array Dimensions

array.

The slight reduction in the performance comes from the fact that the configuration of a DFG needs more rows with small number of interconnections than the same DFG on a fully interconnected array. Especially with benchmarks that are highly sequentialized like the MiBench benchmarks. Thus, DFGs of some loops that can fit into a fully interconnected array need more rows with small number of interconnections, and hence can not be captured in the same array. This effect is eased by connecting the neighbor FUs of the same column directly—as already shown on Figure 4.15—,which reduces the claim on the cross interconnections between the rows. Thus, FUs that operate on the previous value of the destination register and an immediate value does not require a

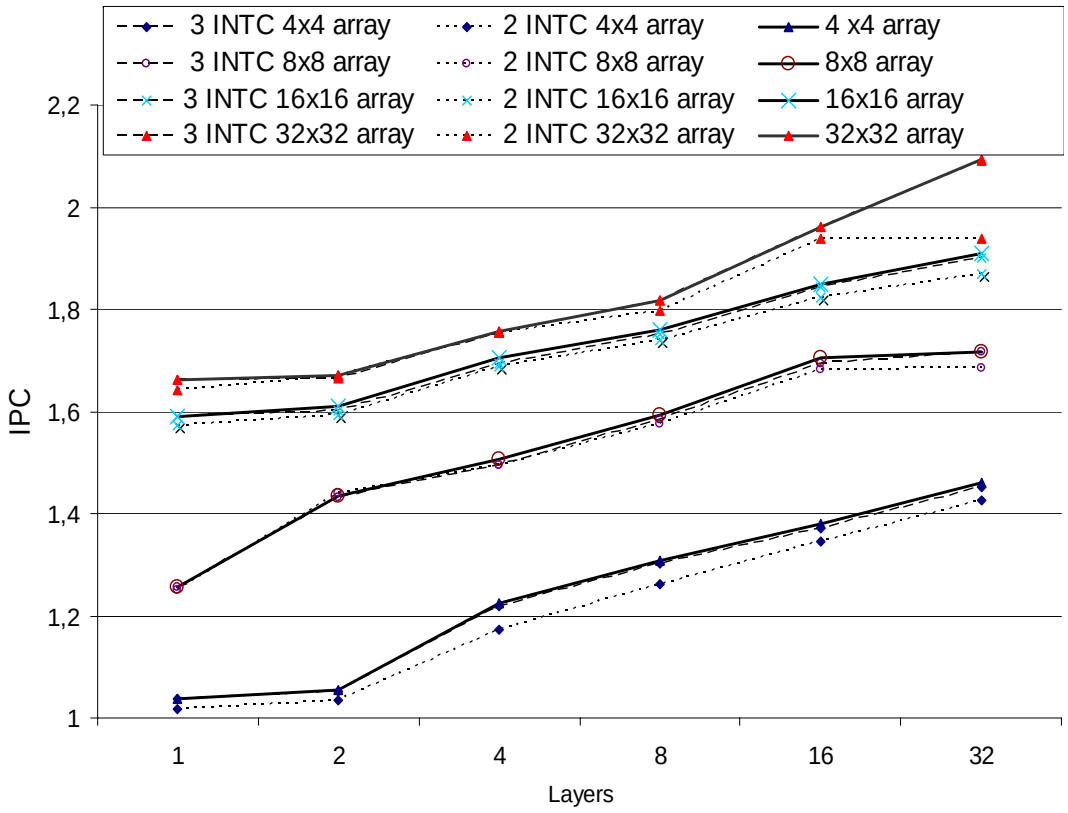


Figure 4.18: GAP average performance with different array dimensions and several configuration layers on MiBench benchmarks. The simulation is done with two and three interconnections compared to a fully interconnected array

dedication of an interconnection, since each FU is directly connected to previous FU on the same column.

Further simulation results are shown on the Figure 4.18 to demonstrate the difference in the performance whilst reducing the number of interconnections. The simulation is done with different array dimensions, several configuration layers, and different number of interconnections. The simulation shows a slight reduction in the performance with two interconnections to reach the performance of fully interconnected array with three interconnections. The only remarkable difference in the performance can be seen with two and three interconnections and 32 layers. As the jump in the performance—with

fully interconnected array and 32 layers—for some Benchmarks like *rijndael* can not be seen with two interconnections, as the computation intensive tile of the program needs more space to fit into the layers. However, the three interconnections allow again more compact configurations to achieve the performance of fully interconnected array.

4.6 Hardware Exploitation of the Functional Units

4.6.1 Related Work

New FPGA reconfigurable devices offer partial reconfiguration capability to enable time-multiplexed reuse of programmable logic resources [36]. Partial reconfiguration allows changing part of the configuration bits of a reconfigurable resource, without modifying the remaining ones. In this manner, multiple functionalities can be assigned to a given hardware resource with little or no impact to overall system performance. With this approach, and the appropriated support [68] it is possible to have several independent tasks running in the same device and to load a new one without interfering with the others. Dynamic partial reconfiguration can also increase the hardware utilization by preserving previous reconfigured tasks and reconfigure the free tiles of the FPGA.

Coarse grained architectures trade off programming flexibility for more efficient reconfigurable hardware. Reducing the programming flexibility has a direct impact in the configuration size and, subsequently, in the configuration latency and in the reconfiguration overhead. However, the interconnection mesh highly impacts the programmability and the utilization of the reconfigurable hardware. Reconfiguring of a part of the resources on the grid requires interconnections to the previous instructions that are already mapped to ensure a correct forwarding of the results.

The processing grid mostly executes basic blocks, which are usually small and occu-

pies a small tile of the processing elements. The grid of the PACT XPP processor [47] is partitioned to four tiles to enable the mapping of different small basic blocks. Similarly , the ADRES [64] as well as the MorphoSys [69] processors underlay a grid divided into four tiles of FUs, where the FUs of each tile are fully connected. In contrast, TRIPS processor [66] executes hyperblocks of code generated by special compiler. The big hyperblocks are generated by applying predicted execution techniques [70] [71] to the branch instructions, which result in mapping useless instructions that are not executed. To gain more utilization of the processing elements in all above stated architectures, many compiler optimizations are implemented like, loop unrolling, pipelining, peeling, flattening, and function-inlining. However, these optimizations are restricted to the interconnection structures on the grid.

In all previous coarse grained architectures, the software tools undertake the scheduling issue, which tries to place the dependent instructions near to each other to avoid long wire communications. This property prevents from highly utilizing the hardware resources. On the other hand the proposed configuration memories are used to enable the mapping of the instructions of the next configuration phase during the execution of previous one. However, our implementation of the configuration layers serves the reusing of previous configurations as in I-cache. Thus, saving more configurations in the grid makes our architecture not only highly utilized but also more efficient.

4.6.2 Dynamic Array Segmentation

The diversity of the code structures in the workloads prevents the optimal usage of the functional units inside the array. Important code tiles like loops can not be captured inside the array due to a lower number of rows or columns as needed. On the first hand, loops that contain instructions with high data dependency require more rows than columns to fit inside the array. On the other hand, loops with high ILP require more columns. And hence, it is impossible to deploy an array of functional units that op-

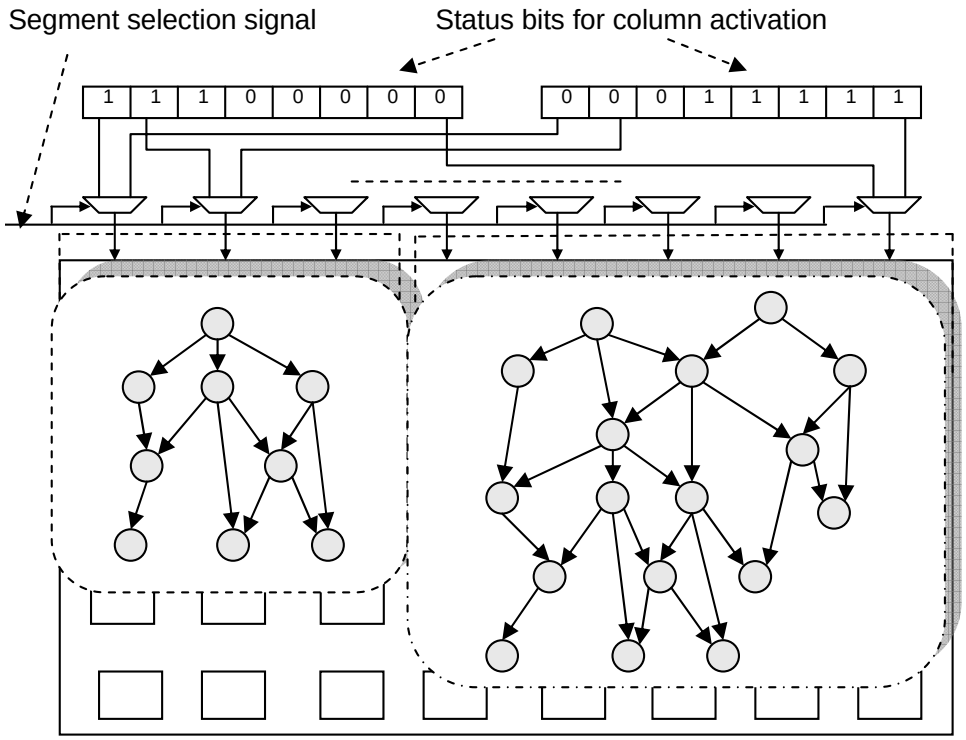


Figure 4.19: Array segmentation with two data flow graphs and Segment selection and activation circuitry

timally response to all possible code structure requirements keeping herewith a high utilization and low costs of the hardware. Even with enhancing the architecture with configuration layers, some resources—columns—stay unused in the two dimensional array i.e. in each configuration phase.

To work around the fixed array dimension, a reallocation of the free regions in the array is necessary. This requires an improvement of the current mapping mechanism in the configuration unit. Figure 4.19 shows two data flow graphs and a possible mapping on an array of functional units with a single configuration layer. A misprediction occurs after executing the instructions of the left-side graph. And hence, according to the previous mapping mechanism—without layers—the control deletes the current configuration and starts mapping the instructions of the right-side data-flow-graph. Assuming an ar-

ray with 8x8 FUs results in 14% utilization for the left-side graph and 26% utilization for the right-side graph. Improving the mapping mechanism to be able to reconfigure the unused columns after finishing a configuration phase keeping thereby the current configurations leads to a much better utilization. This enables mapping both data flow graphs to the array without deleting the first one and increases the utilization of the both segments to 40% in this example. The advantage of this mapping mechanism does not only increase the utilization but also increase the possibility to capture more loops inside the array.

We define the effective utilization as in the following:

$$U_{eff} = \frac{\sum_{i}^{rows} \sum_{j}^{columns} E_(i,j)}{N} \tag{4.1}$$

where $E_(i,j)$ is the number of executed instructions in current segment and N is the number of FUs in the array:

$$N = \sum_{i}^{rows} \sum_{j}^{columns} FU_(i,j) \tag{4.2}$$

and the placement utilization of all instructions in the current segment will be:

$$U_p = \frac{\sum_{i}^{rows} \sum_{j}^{columns} P_(i,j)}{N} \tag{4.3}$$

where $P_(i,j)$ is the number of placed instructions (not matter whether executed or not). The segments placement utilization:

$$U_{pSEG} = \frac{\sum_{i}^{rows} \sum_{j}^{columns} \sum_{s}^{segments} P_(i,j,s)}{N} \tag{4.4}$$

defines the utilization in all segments, where $P_(i, j, s)$ is the number of placed instructions in all segments within a single layer.

Loops that comprise hard to predict branches will be mapped to different segments when a misprediction occurs. After finishing the execution in a segment the control compares the jump address with the first address of all other segments to automatically detect loops as in configuration layers.

The configuration unit changes the borders of each segment dynamically as needed by the current code snippet, which has to be mapped to the array. This dynamicity allows the mapping of different code structures without any change in the underlying hardware of the grid. The only change to be considered in the configuration unit is a status register for each segment. Each bit in the status register activates or deactivates the accompanied column as shown in Figure 4.19. A match between the jump address and the address of a specific segment involves an activation of the accompanied status register and the reserved segment region.

This mapping mechanism is extended to adopt the segmentation of the array and the configuration layers simultaneously, where each layer can dynamically be segmented to several regions. Each layer must be accompanied with a number of column activation registers as in single layer segmentation. The number of column activation registers dedicated to a layer defines the maximum number of segments that the layer can be divided into (N_SEG_i). After finishing the execution in a segment the control of the array starts to compare the address of the instruction to be executed with the first address of each segment as shown in Algorithm 2. The control activates a segment ($activate\_SEG_{(i,j)}$) if the address matches with the first instruction address in it ($SEG\_ADR_{(i,j)}$). Otherwise, the control switches either to the next segment ($allocate(LAY_{curr}, SEG_{curr+1})$) if available or to the next layer ($allocate(LAY_{curr+1}, SEG_0)$). In the case where the current layer is the last one, the control allocates the first layer ($allocate(LAY_0, SEG_0)$). The time needed to check out the match of the address of each segment with that of the

Algorithm 2 The control switch policy for the execution with layers and segmentation option

{;A New address due to a misprediction or array is full and execution is finished in the current segment}
if $new\_address = true$ **then**
  for $i = 0$ to N_LAY **do** {; N_LAY the number of layers}
    for $j = 0$ to N_SEG_i **do** {; N_SEG_i is the number of segments in $layer_i$}
      if $new\_address = SEG\_ADR_{(i,j)}$ **then**
        $activate(SEG_{(i,j)})$
        $stall_{frontend} \leftarrow 1$
        return true
      end if
    end for
  end for
  $Fetch \leftarrow new\_address$
  $activate_{frontend} \leftarrow 1$
  if $SEG_{curr} \leq N\_SEG$ **then**
    $allocate(LAY_{curr}, SEG_{curr+1})$
  else
    if $LAY_{curr} \leq N\_LAY$ **then**
      $allocate(LAY_{curr+1}, SEG_0)$
    else
      $allocate(LAY_0, SEG_0)$
    end if
  end if
end if

instruction to be executed occurs in parallel to result write back. And hence, no extra time has to be spent in the control of the array.

4.6.3 Evaluation

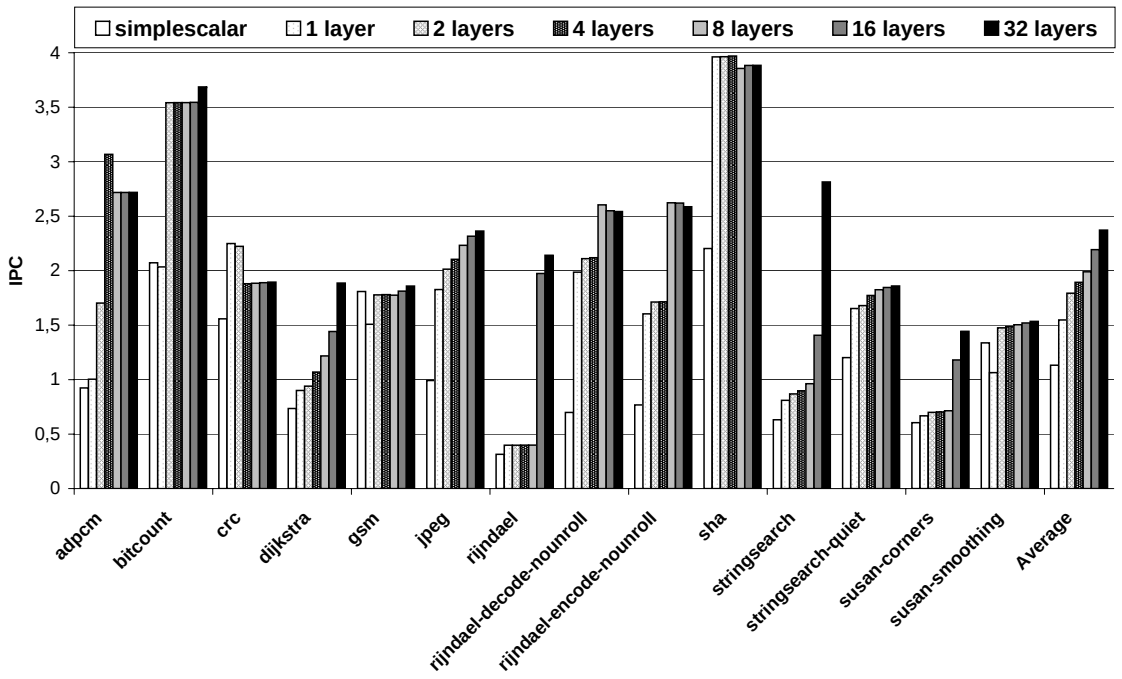


Figure 4.20: GAP performance with 32x32 array and several configuration layers with segmentation option in comparison to SimpleScalar

Several aspects of the execution inside the array of GAP contribute to its performance. All benchmarks comprising one or more loops benefit from the loop acceleration in the GAP's array. The array exploits high ILP and memory access parallelism in the loops and functions. Moreover, it achieves a speed-up by executing the already mapped loop body asynchronously. Multiple configuration layers reduce the penalty of misprediction inside the loops and the functions significantly, since a misprediction switches the control to another configuration layer without replacing the current configurations. The

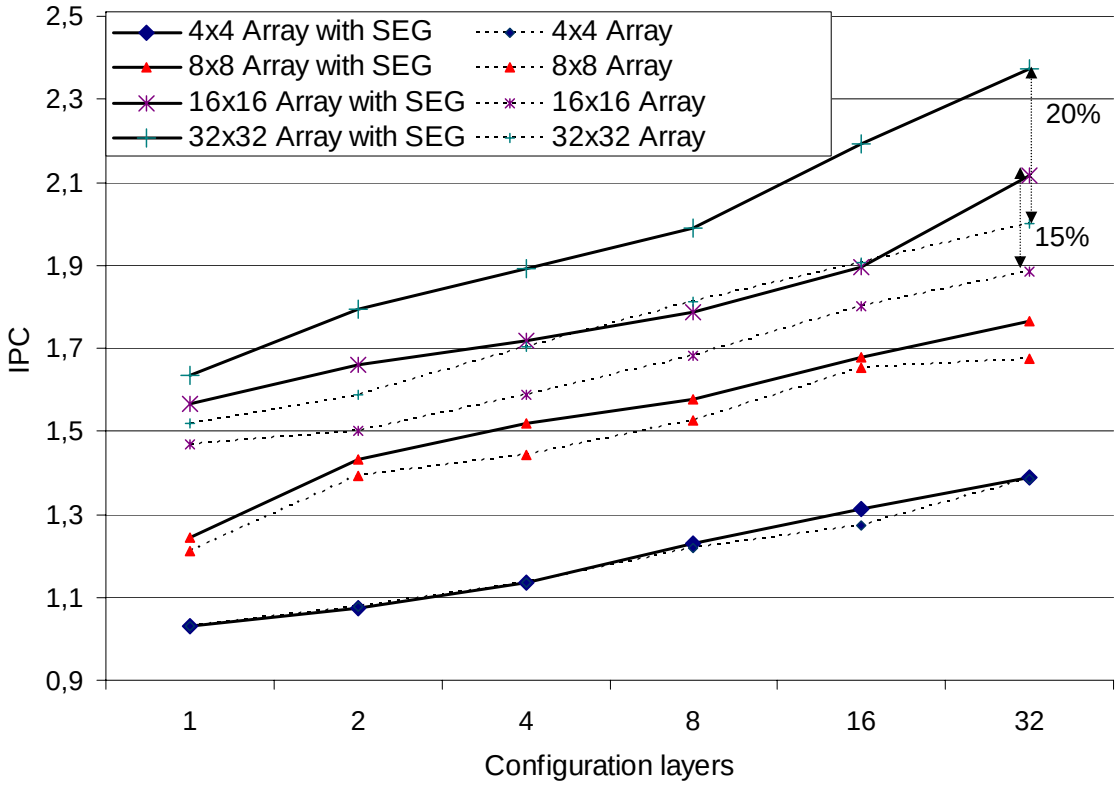


Figure 4.21: GAP average performance with several configuration layers and array segmentation vs. non segmentation

separation of each layer into many segments can be also considered as extra layers. Hence, many aspects of the simulated benchmarks with segments and layers together are similar to that of simulating with layers.

Figure 4.20 shows the performance of GAP using the segmentation option with several configuration layers in comparison to the results of the out-of-order SimpleScalar simulator. All previous optimizations in this section are also applied to the simulated GAP in this section like: one multiplier/divider, several dimensions of the array, configuration layers, and two interconnections between the rows. The more layers are deployed the more loops can be captured in segments of the array, and hence, the better

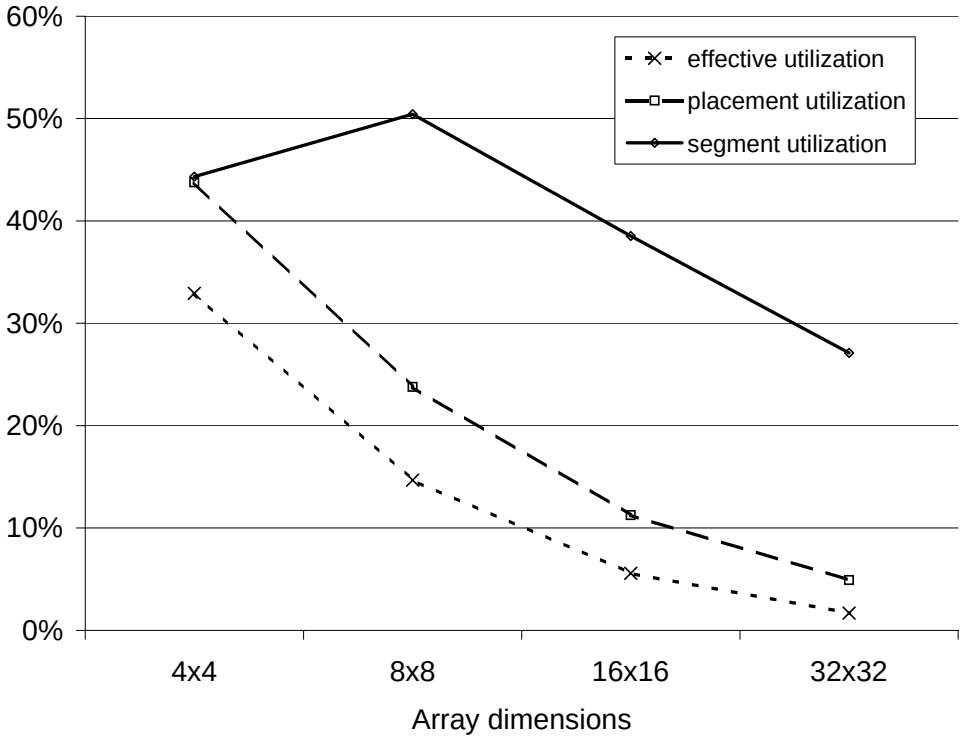


Figure 4.22: Average effective utilization, placement utilization and segment utilization on GAP with one layer and different array sizes

is the average performance that can be achieved. The highest performance of GAP—reached with 32 layers—overcomes SimpleScalar with a factor of 2.2. For some benchmarks however, a small degradation in the performance occurs with a higher number of layers, e. g. for *adpcm, crc* and `sha` (16 vs. 32 layers). This is incurred by the control flow in the loops and the functions as explained previously in Section 4.4.

A detailed discussion of the high improvement of the performance of *rijndael* is stated in Section 4.4.4. A jump in the performance occurs as the layers technique allows to save the huge computation intensive part in the layers and avoids a lot of cache accesses and misses. Moreover, the long configurations are executed faster, since the execution is asynchronous. With the segmentation option, the jump in the performance occurs with less than 16 layers, whereas it occurs only with 32 layers without segmentation.

Hence, the segmentation makes the configurations more compact by distributing them in the second dimension—two dimensional array—instead of switching always to the third dimension—the configuration layers—. The *stringsearch* is also showing a similar improvement as *rijndael* with 32 layers due to the same reason as with *rijndael*, which can not be seen with an array without segmentation.

Figure 4.21 shows the simulation of GAP with different array dimensions for a better comparison between the performance with segmentation vs. non segmentation. A 20% better performance with 32x32 array and segmentation can be achieved compared to a non segmented array. A 16x16 FU-array shows also 15% performance improvement over the same array dimensions without segmentation. However, a 4x4 array does not show any improvement, since the dimensions are too small to separate the array into segments. This also can be seen on the Figure 4.22, where the segment utilization does not differ from placement utilization of a single segment for 4x4 array. The segment utilization improves with a 8x8 array to occupy the half of the FUs in the array for the simulated benchmarks. With bigger array dimensions, the segment utilization drops to occupy 27% of the FUs with 32x32 array. However, the segmentation on 16x16 and 32x32 array results in 30% more utilization than placement utilization without segmentation.

Chapter 5

Data Cache Hierarchy

5.1 Introduction

The design of cache/memory hierarchy is an important issue for emerging processor architectures, since the memory bottleneck is the most hampering factor of the performance [72] [73]. An effective organization of cache/memory hierarchy can absorb the delay incurred by the memory bottleneck to its minimum and boost the performance of the whole design. A key factor of the whole performance is the memory access structure. A determining question herewith is how to parallelize and schedule the memory accesses. Different hardware and software techniques are researched to hide and tolerate the memory latency. Nonblocking cache technology offers the underlying structure for access parallelization and miss overlapping. These caches allow multiple-pending cache-misses in order to overlap the latencies of different next-level accesses.

The memory-runahead execution technique [74] [75] introduces a method to avoid stalling the pipeline during a L2 D-cache miss by speculatively executing the waiting instructions. Thereby, the pipeline exploits more cache misses simultaneously. However, the memory-runahead technique adds hardware overhead and needs checkpoints

support and a rollback mechanism, since the execution during the runahead mode is speculative. Moreover, the speculative execution adds more energy consumption by executing unnecessary instructions. Other techniques for predicting the address of next memory access based on prefetchers have shown more or less success. Also many different approaches based on data correlation have shown an improvement to reduce the D-cache misses. Simultaneous multithreaded architectures (SMT) can also overlap several misses from different threads [76] [77]. However, the scheduling of the memory accesses requires a compiler analysis and optimization of the statically generated code. Moreover, even an acceleration in the execution of parallel threads can be achieved, single threaded applications can not benefit from such techniques.

Each row in the reconfigurable fabric of GAP processor is associated with a load/store unit to offer an access to an accompanied D-cache and provide the executing instructions inside the array with the necessary data from the memory. This expands the D-cache of GAP processor to effectively response to multiple requests from different load/store units. Herewith, a special data cache hierarchy is introduced to offer a hardware based exploitation methodology for parallel cache/memory accesses.

This chapter presents and evaluates the data cache hierarchy for the GAP processor. It shows how this organization responses to multiple requests and meets the requirements of the special design of the GAP processor.

5.2 Data Cache in Coarse Grained and Clustered Architectures

Coarse grained reconfigurable processor designers have been mostly tried to avoid deploying a D-cache access scheme beside the grid execution core. Usually the execution inside the array of the processing elements is out-of-order. Therefore, the mem-

ory disambiguation becomes a non trivial problem especially with distributed D-cache schemes [61, 5]. The TRIPS microprocessor is implemented with a distributed D-cache and a support mechanism of the out-of-order execution nature inside the reconfigurable core. The sequential semantic of the memory system is kept using complex mechanism enhanced by a dependence predictor and banked-by-address cache organization [78]. The RAW architecture comprises complex processing elements like many-core systems with shared D-cache for each row of the processing elements. A compiler techniques based on the concept of memory-bank disambiguation for distributed bank architectures are used to statically determine the referenced memory bank [79].

Clustered architectures are conceptually different from coarse grained reconfigurable processors, as the back-end consists of several clusters comprising different issue queues, register files, and functional units. However, the cache organization could exhibit similar characteristics. A clustered architecture with a cache/memory access scheme using a centralized D-cache as well as load store queue (LSQ) can be found in [80]. The handling of cache/memory accesses in the case of centralized D-cache is similar to that of conventional out-of-order processors with the difference that finished loads must be forwarded to the requesting cluster after resolving previous stores. Heuristics in the steering and renaming unit—are focused on in [81]—to issue load/store instructions to a specific cluster in order to avoid the communication overhead. Following works have presented a distributed cache models [82, 83, 84], where several strategies has been researched to keep the memory disambiguation in the distributed cache design. One of them is based on banking the D-cache in several clusters and steering the load/store instruction to the cluster with the data holding cache [85]. Other work has explored the attraction policy—where the data comes from memory goes only to the requesting caches—with disambiguation buffer and bus [86, 87].

The D-cache organization of the GAP differs from previous schemes of out-of-order coarse grained reconfigurable processors by simply handling the cache accesses and keeping the consistency without any compiler optimization or extra dependency predict-

ing mechanisms. The in-order mapping of the instructions and top-down arrangement of load/store instructions in the load/store units of GAP processor is a crucial issue to allow a simple memory disambiguation. Therefore, the design scheme of the D-cache of the GAP can not be generalized or applied directly to other coarse grained reconfigurable architectures without a similar modification inside the grid as in GAP. The work presented in this chapter differs also basically from previous works in clustered architectures. Our observed class of architectures deploys an array of functional units instead of clusters. Moreover, the mapping of instructions does not take into account any considerations except the data dependencies of the operations to ensure the precise overhanding of results.

5.3 First Level Data Cache for GAP

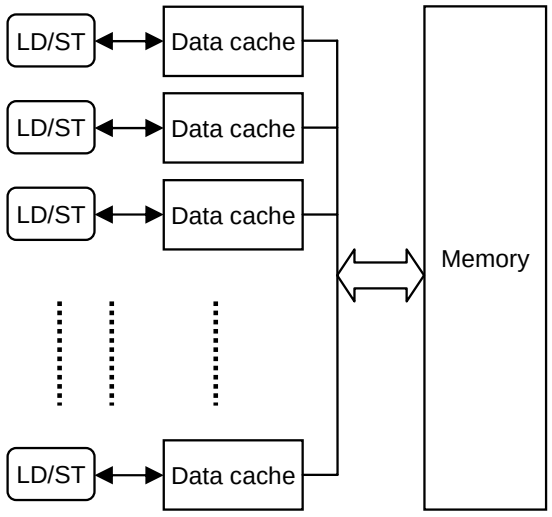


Figure 5.1: Cache/memory organization with access structures

A centralized L1 D-cache can not provide enough bandwidth commensurate to the aggressive execution inside the array of FUs along the line with low latency. This

opens the way to distribute the D-cache and arises many issues for memory disambiguation. Data access parallelization in GAP is introduced by implementing several LD/ST units that can access their own relatively small D-caches in parallel as shown in Figure 5.1. The memory accessing subsystem allows servicing several loads in parallel. This organization ensures high memory access parallelism for architectures with intensive computation core, like coarse grained reconfigurable processors. The parallelization of memory accesses can also exploits more ILP and boost the performance of data intensive applications.

The D-caches accompanied to the LD/ST units envision a first level data cache distributed vertically beside the execution core. The D-caches in the first level are kept different in order to avoid duplicating the data in different D-caches and reducing the conflict misses. Hence, maximally one copy of the data is available in all D-caches in the first level. This organization enables saving more sets in the cache and acquires less communication, complexity, and coherence efforts. Nevertheless, the communication between the D-caches are solved by a common bus between the caches and bus interface units (BIU), where each D-cache is equipped by a BIU that can snoop on the bus and answer the request if it detects a valid copy of the data.

As the execution of the operations in the array is out-of-order, loads/stores can request the data from memory as soon as the execution of previous operations—on which they are dependent—has been completed. However, an out-of-order access and completion can result in WAW, WAR, and RAW hazards. In order to avoid this kind of hazards another special token signal (memory token signal) shown in Figure 5.2 is introduced to indicate a possible data request from the cache/memory subsystem or a possible completion. As shown in the figure, the load access can start when the token signal has arrived and regardless of the memory token signal. However, the memory tokens signal can only proceed to the down laying LD/ST units if the token signal has arrived. In contrast, the store access can start only when both token signals are available. More details about the access scheme with the memory token signal and different sequences

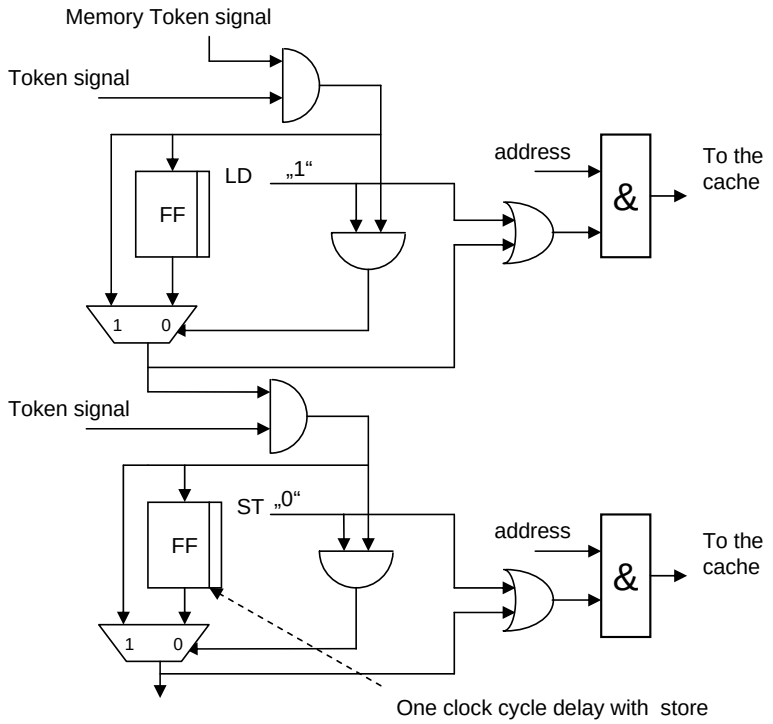


Figure 5.2: Memory token signal to solve access conflicts

of loads and stores are discussed in the following:

Store access: The store access updates the value in the associated D-cache—if a hit—or in other D-cache containing a copy of the data—in the case of a miss in the associated D-cache—and in the memory (write-through).

- Store after load: A store access starts if all previous loads/stores have sent their requests and the successive loads/stores must wait until the store finishes writing. The load/store instructions are placed in-order and from top to down in the LD/ST units. Hence, an in-order request and out-of-order completion in the case of store after load is possible. The in-order access of the store is realized by the memory token signal that flows through the load/store units and indicates a possible data request from the cache/memory subsystem. The memory token signal ensures that all previous loads have sent their requests before the store access starts. However,

the completion of the requests stays out-of-order, where the store access can write a previous requested value—by a previous load—and result in WAR hazard. To avoid this, the BIUs of the upper D-caches take into account the address of the miss request—if exists—and update the value in the accompanied D-cache after solving the load miss. The BIU can recognize a value update by a successive store by sending the priority with the data.

- Store after store: As mentioned previously the access of the store is controlled by the token signal, which ensures the in-order access of the store. By a store access the token signal must be hold for one clock cycle to ensure writing/reading the correct data by other accesses and avoiding conflicts.

Store queue: A store miss can be solely resolved by deploying a store queue (SQ) without suffering long time latencies when the data is requested again. A centralized SQ can be easily managed, but it incurs undesirable latencies on the bus. The delay incurred by the communication on the bus herewith is significantly less than only storing the data in the second level or in the memory and acquire it again. The latencies differ based on the physical wire distance from the access requesting LD/ST unit, when a value must be written or read to/out of the SQ. In contrast, a distributed SQ spars the communication overhead. Therefore, the implemented SQ is also distributed as with the D-caches. Each D-cache is accompanied by a SQ, where all SQs contain the same copy of data. With the help of a dummy entry in each SQ, the data can be stored temporarily and then updated in the next clock cycle in other D-bank or SQ.

Load access: The load access sends the request to its private D-cache, and receives the data back if it was a hit. If the access was a miss in the dedicated cache the request must be sent to the other D-caches—with short delay in the case of a hit—or to the memory in the case of a miss in the first level.

- Load after store: To tolerate the delay of a load after a store without deploying a load queue, the load can access the D-cache/memory even before starting the

request of the store. As shown in Figure 5.2, the load access can start when the token signal is available. However, the load completion must wait until the store has been finished to avoid RAW hazard. Thus, the access in this case is out-of-order request and in-order completion. If the load was a miss and the store is writing the same address of the requested load value, the request sent to the memory—by the load access—must be canceled. Finally, regardless whether the miss or the store is resolved earlier, the correct value coming either from memory or from the store instruction will be read.

- Load after load: The loads can start their request independent of the accessing state of other loads. Several loads can start the access simultaneously (or out-of-order), where the completion is out-of-order.

From previous considered cases, the load instruction is always allowed to request the data as soon as the input values are available. However, the access completion must be controlled by the memory token signal. Moreover, the value-update keeps the D-caches consistent with small restriction on the cache level parallelism in the store case and without the need for complex data-cache coherence protocols.

The presented data cache access scheme offers a high accessing parallelism for data intensive applications by servicing several loads in parallel, and leads to better ILP exploitation for basic blocks in data flow as well as control flow dominated applications. More details about the D-cache organization and the access behavior in several cases are stated in the following sections.

5.3.1 A Hit in the Dedicated D-Cache

Figure 5.3 shows a demonstrative data flow graph (DFG) representing a loop body that can be configured into the array of FUs in the GAP processor. After detecting the loop in the grid, other stages in the processor stall and the execution continues inside the array of

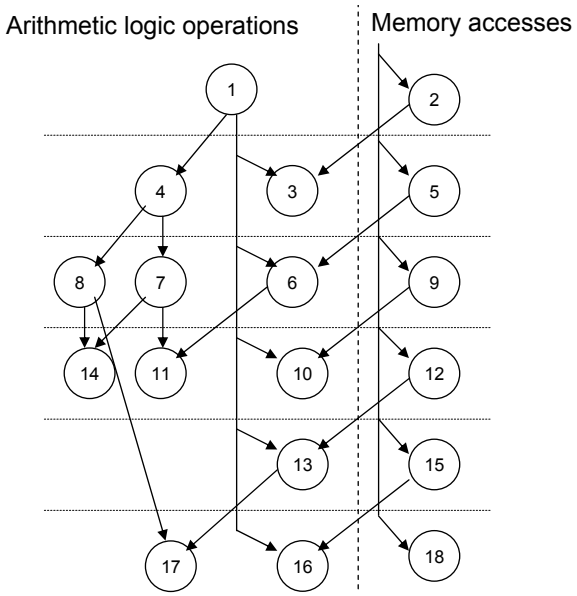


Figure 5.3: Increasing the ILP vertically by servicing the memory accesses earlier

FUs. The best scenario execution on the GAP occurs when each load finds the required data in the accompanied D-cache. An advantage over the out-of-order processor is the memory access parallelization of many load/store instructions. In this example all loads/stores start their access independent of the other arithmetic instructions. Thus, even the memory access instructions are not favored during the first mapping to the FUs they can start the access as soon as possible in the following iterations, whilst an out-of-order processor issues the instructions in their order in the absence of data dependency. This incurs possibly an avoidable delay, whereas sending the requests to the memory as early as possible and executing other instructions meanwhile lead to better execution times.

Another advantage of the parallel memory accesses is the increasing of the vertical instruction level parallelism. The vertical parallelism is exploited by servicing several loads simultaneously, where instructions 3, 6, 10, 13, and 16 can start the execution as soon as the load instructions have been serviced. The execution of this DFG on an out-of-order processor takes at least six clock cycles dominated by the underlying dependen-

cies. However, mapping this block of instructions into an array of FUs saving herewith the configuration in the array increases the ILP of the block. The reader can imagine a 90° rotation of the data flow graph and consider the parallelism after that. In this example, the ILP increase to seven instructions in first clock cycle (1,2,5,9,12,15,18), and to six instructions in the second clock cycle (3,4,6,10,13,16), etc..

5.3.2 A Hit in Another D-cache in the First Level

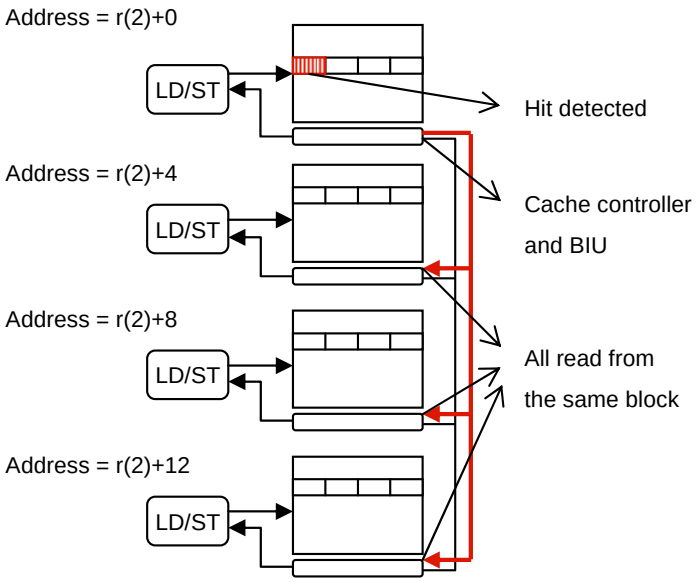


Figure 5.4: Data hit in another Data cache in the first level with 4-way set associative data cache

A non optimistic scenario of the GAP D-cache access occurs if the requested value is not available in the associated D-cache, or if many loads are trying to read from the same cache block in the same clock cycle. In the case of a load miss in the dedicated D-cache the request must be sent on the bus to other D-caches in the first level. Each D-cache is equipped by a BIU that can snoop on the bus and answer the request if it detects a valid copy of the data (after checking the tag container of the D-cache).

Figure 5.4 shows four load instructions requesting data of a same block in the D-cache. The first access successes and returns a hit in the accompanied cache, whereas other three accesses return a miss and send the address on the bus. Only one BIU can send the address on the bus, where others keep snooping. The first D-cache with BIU detects a valid copy of the required data and sends the whole block on the bus. Each requesting BIU can detects the valid data on the bus and either saves it in the own D-cache or directly forward the data to the load/store unit. In our Scheme the BIU ignores saving the block in its own D-cache as duplicating blocks can increase the cache pollution in the first level. However, a load unit that tries to read data incrementally—in the next requests—from the same block can suffer many delay times if the BIU discarded the block of the first transfer. A solution around this problem is to save the block in the cache buffer and service the next requests directly by the cache controller even without accessing the dedicated cache. Consequently, the D-caches are kept different without data overlap in their cache lines and as less as possible communications on the bus.

5.3.3 The Miss Handling in the First Level D-Cache

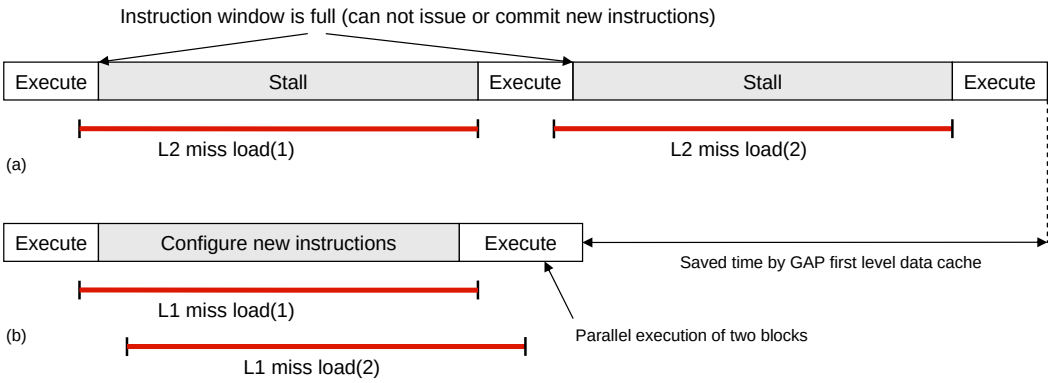


Figure 5.5: Execution timelines of a) out-of-order processor with small instruction window and b) GAP Processor

If a request was sent to the memory according to a miss in the first level, other BIUs

observe the requested address. Another cache miss in a different D-cache and the same block address can be recognized by the BIU to receive a copy of the read data from the memory when the first miss is resolved. Thus, assuming that the first access in Figure 5.4 incurs a miss, other load accesses wait until the first access is resolved and forward the data simultaneously to their load/store units.

In the case of a store after load miss, where the store is writing the same address of the load, the BIU keeps a copy of the data in a dummy entry of the SQ and updates it in the D-bank of the load miss only after finishing the load access. The dummy entry in the SQ must be deleted after the updating in other D-cache to keep only one copy of the data in the first level.

Conventional out-of-order processors try to tolerate long memory latencies by using instruction buffers. In the case of a L2 cache miss the processor tries to exploit more parallelism by searching the instruction buffer for data independent instructions. With a small buffer it comes to long stalls as the instruction window becomes soon full and following instructions cannot be committed, whereas a big buffer involves large, slow and energy hungry structures. Figure 5.5 shows the execution timelines of a conventional out-of-order processor with a small instruction window in comparison to the execution timelines of the GAP processor. The conventional processor suffers long time stalls, since it can not exploit much instruction parallelism and the small instruction window becomes soon full, where following instructions cannot be committed [75]. However, the GAP processor can overcome this bottleneck by mapping new dependent and independent instructions to the array irrelative to the D-cache access state. As soon as a load/store is ready to be executed the request will be sent to the corresponding D-cache exploiting thereby more possible D-cache misses, where the two misses shown on the figure are data-independent. Overlapping the two misses saves significant execution time especially with memory access of hundreds cycles access delay. Another cache miss can also be exploited as long as the array is not full.

The execution in the GAP processor is limited to the physical dimensions of the array as mentioned. The execution in a configuration phase ends only when all mapped instructions have been executed (or all the instruction before the branch with a misprediction). Even with implementing an array with several configuration layers the execution nature in a single configuration phase does not change. A switch between the layers occurs after finishing the execution in current layer. This can be considered as a drawback of the execution in the GAP's array, especially when the array is implemented with small number of rows. A load miss in the last row of the array incurs long time latency without to be able to execute other instructions that are possibly already mapped to different layer. A partial solution around this problem is offered in Section 5.4.2 by developing an address prediction mechanism for the data cache access instructions inside the loops.

5.4 Anticipating the D-Cache Misses

5.4.1 Second Level Data Cache

Taking a look at the Figure 5.6 reveals that more than two misses can outstand simultaneously from the D-caches in the first level. The resulting latencies of cache misses could be on an order of hundreds of clock cycles and apply an enormous negative effect on the performance. Therefore, deploying a nonblocking second level D-cache to anticipate the first level misses can reduce the number of misses going to the memory and avoid long time latencies. The simulated nonblocking D-cache can service up to 8 requests in the presence of a miss and does not require any table to keep track of dependencies as in out-of-order processors, since possible inconsistencies are already solved in first level by the presented D-cache organization.

The second level cache in this work is devoted to data contents to analyze its effect

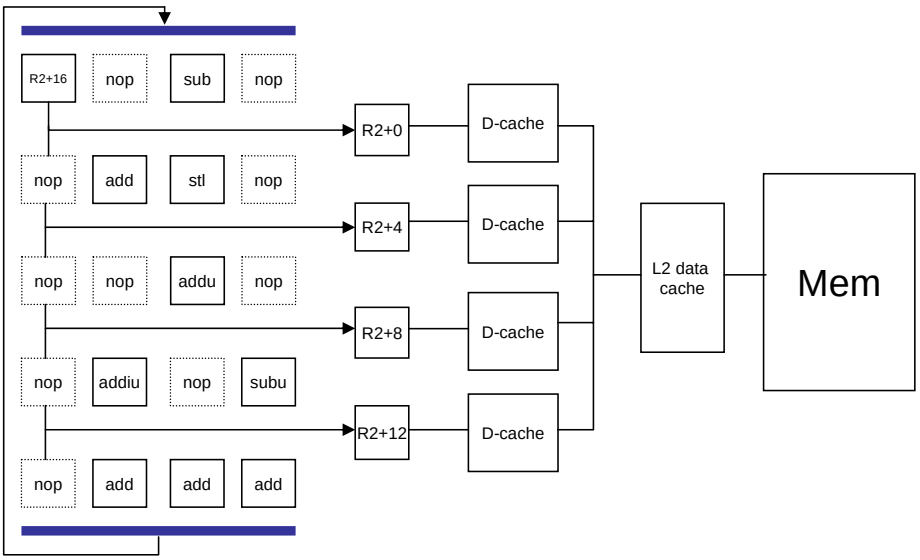


Figure 5.6: An array example with the data cache/memory access scheme

on the data access scheme. The management and the access handling of the cache do not differ from conventional second level caches in out-of-order processors except the tracking of the requests. The second level cache handles the requests from the first level sequentially and sends back the required data on the same bus. If the request was a miss in the second level, a memory management unit (MMU) translates the logical address to a physical address and sends it to the memory.

5.4.2 Address Prediction

The execution in the grid alu processor is restricted to the array limitations. This can considered to be a drawback for the memory access parallelization. The example shown in Figure 5.6 is taken from *sha* benchmark and demonstrates a loop placement in the array with four load instructions reading from the same block (the same register is used with an immediate value shifting for addressing the data). The upper load instruction incurs a miss each time execution and leads to execution timelines on GAP similar to

upper part of the Figure 5.5, since the next D-cache/memory access starts after executing all instructions of the current iteration. To overcome these slots of undesirable wait, a small change in the control of the memory access units is applied to start prefetching the next address even before the execution arrives the same load instruction again (before finishing current iteration). This change can predict the next data address in the memory and does not require any other structures to implement it. For this purpose each load/store unit must be able to save the last address in order to extract the probable distance between the two accessed data in the cache/memory. In the case of executing a loop in the array the concerned load/store unit subtracts current address from the old one to extract the distance between the two loads. Afterwards the memory access unit starts prefetching the current address plus the distance (the immediate value) as listed in following szenario:

```
at time t     :  send address(t)
at time t+n   :  send address(t+n)
at time t+n+1 :  imm = address(t+n) - address(t)
;While loop repeat:
at time t+x   :   send address(t+x) + imm
```

Consequently, the distance must be calculated only once each loop, and then added to the current address each time the execution arrive the holding load/store unit. Improving the last method to prefetch two addresses each time execution performs better, especially with small loops and long memory latencies (where the execution arrives the load very soon again). Extending the LD/ST units to support address prediction in the example in Figure 5.6 improves the execution timelines as in the lower part of the Figure 5.5.

Parameter	SimpleScalar	GAP
L1 I-Cache	128:64:1	128:64:1
L1 D-Cache	1KB x GAP-Rows	1KB for each L/S unit
Bus delay between the D-caches	-	1 clock cycle for two rows distance
L1 cache ports	2	1
L2 D-cache	16KB	16KB
TLB	-	-
Data cache queue	LSQ 128 entries	SQ 128 entries
Cache latency	2	2
Memory bus width	16 bytes	16 bytes
Memory ports	2	2
Memory latency	24	24
replacement strategy	LRU	LRU
associativity	2-way	2-way
Data cache block size	16 bytes	16 bytes

Table 5.1: List of memory parameters for GAP and SimpleScalar

5.5 Evaluation

The memory configurations of GAP and SimpleScalar simulators are listed in Table 5.1. We kept the D-caches of GAP in the first level small (1KB each) according to the physical dimensions with the distribution beside the grid. The D-cache size for both simulators is the same, where simulating GAP with a four row array—for example—yields a 4 KB data cache in the first level and is compared to SimpleScalar with 4 KB data cache.

Figure 5.7 shows the performance of GAP with several D-cache organizations and different array sizes (i.e. different D-cache sizes) without layers in comparison to SimpleScalar. The SimpleScalar does not deploy an array as in GAP. However, the improvement in the performance of the SimpleScalar comes from increasing the size of

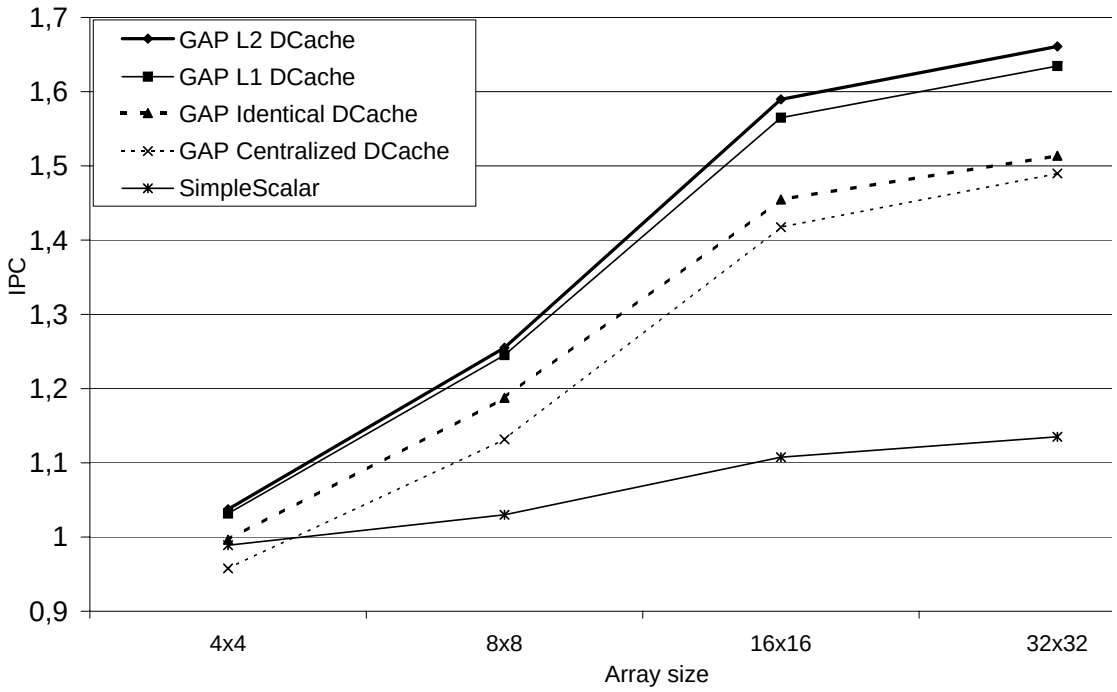


Figure 5.7: Average IPC of GAP with several data cache organizations and different array sizes compared to SimpleScalar with different data cache sizes

the D-cache with the more number of rows in the simulated GAP.

Centralized data cache organization deals with requests as in out-of-order processor, where a LSQ structure solves the store-after-load and load-after-store cases. The scheme of identical D-caches means that each D-cache in the first level is kept similar to all other D-caches accompanied to other LD/ST units. Thus, there is no communication between the caches by load access, since all are identical. The only coherence restriction in this case occurs by writing a value to a D-cache, where all other D-cache must keep snooping on the bus to update their values. In the case of a D-cache miss in the associated D-cache the request goes directly to the memory. The coming value from memory must then be sent to all D-caches in the first level.

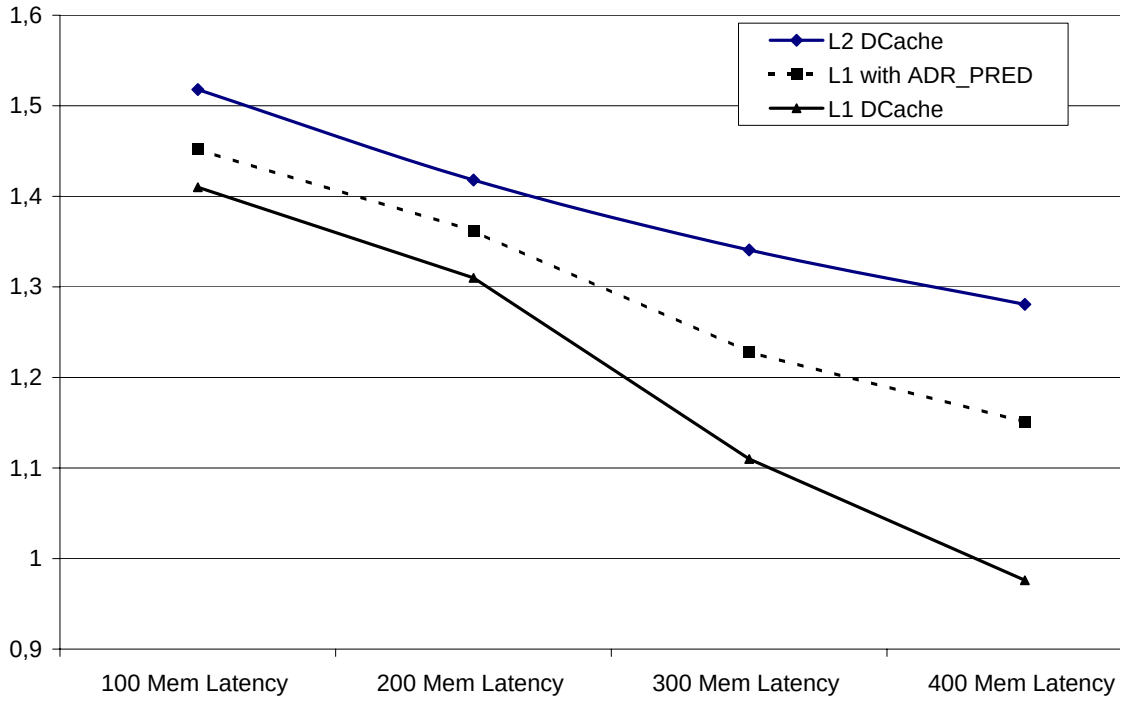


Figure 5.8: Average IPC of GAP with different memory latencies and first level cache without/with address prediction in comparison to second level D-cache

The GAP with L1 D-cache overcomes the SimpleScalar with the factor of maximally 1.5. The scheme presented in this work for D-cache organization overcomes the performance of centralized as well as identical D-caches by about 10%. The difference in the performance between our scheme and the centralized scheme increases with array sizes (i.e. number of caches in the first level), since the distance between the centralized D-cache and the LD/ST units increase with more rows in the array (one clock cycle for two rows distance).

Our organization envisions a first level D-cache constituted by a high number of blocks (of all D-caches) and a number of sets equal to the single D-cache sets. However, a drawback of this organization is the replacement strategy of the blocks in the D-caches, since each D-cache is equipped by a private LRU table. By replacing a block in a D-

cache, the least recently used block will be evicted, whereas possibly less recently used blocks of the same set in other D-caches stays untouched. Hence, increasing the size of the D-caches in our scheme reduce the effect of the conflict cache misses—caused by evicting a block from the cache—more than centralized D-cache, since the replacement strategy in the centralized D-cache is better than that of distributed D-caches.

With L2 D-cache a better performance is also achieved. However, the change in the performance with second level D-cache is small as shown in the same figure. This goes back to the benchmarks and the configuration characteristics, where some of the tested benchmarks are operating on small data sets. The benchmarks with small data sets hide the effect of the nonblocking cache of other relatively data intensive benchmarks. To show the effect of the second level D-cache we increased the memory latency for up to 400 clock cycles. Figure 5.8 shows the GAP performance with L1 D-cache with/without address prediction and second level D-cache. The GAP is simulated based on a 16x16 FUs-array (i.e. 16 KB D-cache in the first level), 16 KB L2 D-cache, and different memory latencies. The simulated address prediction prefetches two blocks in the case of a miss in the first level, where the prediction mechanism is operating only inside the loops as explained earlier. This allows a very precise prediction and avoids increasing the cache pollution with unrequested blocks (which occurs mostly outside the loops).

The effect of long memory latencies incurred by cache misses in the first level are absorbed by the nonblocking D-cache in the second level as shown in the figure. The second level D-cache has increased the performance of GAP by about 30% with high memory latency (400 clock cycles) and about 6% with relatively low memory latency (100 clock cycles). Address prediction also increases the performance with first level D-cache for about 15% with 400 clock cycle memory latency and about 3% with 100 clock cycles memory latency.

Chapter 6

Conclusion and Future Work

In this chapter, we give an overview and outlook over the presented work. A conclusion of the implementation and optimization of coarse grained reconfigurable architectures with emphasis on the GAP architecture is offered. Finally, we discuss further optimizations and studies that can be done in the future.

6.1 Conclusion

This work addresses many issues in the coarse grained reconfigurable architectures. A solution of a main challenging issue of accelerating the mapping of instructions into the grid has been proposed in the GAP processor. With the GAP, we introduced a method for configuring the instructions in the hardware and mapping them into to the grid without any software interference. Thereby, the GAP with its hardware-based reconfiguration unit absorbs the delay resulting from the cooperation between the hardware and software.

The great advantage of the GAP architecture is the performance improvement for sequential workloads. The runtime reconfiguration of an array of functional units grants

the processor a big dynamicity and the ability to effectively execute control flow as well as data-oriented instruction streams. This dynamicity allows the deployment of a simple in-order processor-frontend without the need of a controlling processor, a special ISA, and special software or new compiler to analyze the data flow graphs and extract the configurations as in similar reconfigurable architectures. Moreover, saving the generated configurations in layers and the asynchronous execution in the array further improve the performance. ILP is exploited herewith by many factors, including: First, the array can execute an already mapped loop body. Second, layers of configurations also increase the number of instructions that are ready to execute inside the array (nested loops and functions). Third, many load/store units expand the parallelism of the basic blocks vertically and accelerates the memory accesses for data intensive applications. Additionally, mapping dependent instructions simultaneously shortens the execution time of the critical path to reach the minimum in already mapped loops.

The optimization done in the hardware was directed towards reconfiguring an array with small dimensions. More efforts to reduce the hardware costs have led to implement a special reconfigurable multiplier/divider as shared unit, whereas the FUs in the array are responsible for the execution of simple arithmetic/logic instructions. Further improvement of the performance with small array dimensions has been achieved by dedicating more configuration memories beside the FUs (configuration layers). The proposed configuration layer management differs principally from that of reconfigurable architectures by managing the layers as an effective cache instead of using it for holding the expected new configurations. The mapping strategy is also improved by reallocating the free regions in the array (dynamic array segmentation) to increase the array utilization and capturing more loops and functions. This allows also reducing the array size and the number of layers without to hamper the performance gain. Another hardware optimization is presented to reduce the number of interconnections and the size of controlling elements (multiplexers/demultiplexers) on the grid. We have shown that an array with three interconnections between the rows achieves the performance of

a fully interconnected array, where the reduction in the performance is marginal with only two interconnections. All these discussed optimizations can also be generalized in the coarse grained reconfigurable processor with a special care about the characteristics of each design.

A distributed D-cache hierarchy with hardware-based memory-disambiguation is presented to respond to the intensive execution inside the grid. The access scheme ensures a simple memory disambiguation controlled by the in-order arrangement of load/store instructions in the LD/ST units and a special memory token signal. The D-cache is also enhanced by an address prediction mechanism and a second level nonblocking cache. Our organization scheme has shown a better performance over the centralized and identical D-cache organizations in the first level. The impact of address prediction and second level D-cache is also depicted with relatively high memory latencies.

6.2 Future Work

6.2.1 Hardware Cost

The most hardware-consuming part of the GAP and related coarse grained reconfigurable architectures is the grid including the processing elements and the interconnections. We have discussed several optimizations in Chapter 4 to reduce the hardware costs without to hamper the performance. Anyhow, the more FUs in the array the better is the performance that can be achieved.

Different characteristics of the GAP processor make it promising to start a hardware cost study. These characteristics are:

- The processing elements on the grid of the GAP processor are very simple (only ALUs and interconnections) in comparison to the ones of related architectures.

- The FUs in the GAP processor are specialized, such that only one multiplier/divider is implemented, whereas other FUs in the array are simplified to execute the simple arithmetic/logic operations.

- The Itanium II die photo reveals that less than two percent of the die area is dedicated to its 6-way issue integer execution core [67]. Thus, on the first hand the array of FUs does not really sacrifice a large die area in the contemporary processors. On the other hand our simulations show that an array with 4x4 FUs and 32 layers achieves a speed up of 1.5 in comparison to SimpleScalar.

- The GAP processor fetches, decodes and configures a stream of in-order instructions, which make the frontend of the design very simple and does not imply the use of large and power-hungry instruction window, register renaming and reorder buffer—with the needed hardware for searching instructions that can be executed—used in out-of-order processors. GAP saves herewith a significant amount of resources to compensate the use of more functional units and the configuration unit.

All of these observations can only be proven by a hardware implementation of the GAP processor. Concrete and precise information about the hardware costs of the GAP processor can be gained during the ASIC synthesis. Additionally, a very important and open issue to be studied is the critical path analysis in the configuration stage for row selection as well as timing circuits.

6.2.2 Power consumption

The power consumption is also an important issue in contemporary processors, such that it plays an important role beside the performance and the hardware costs to offer an effective design. In this section, many aspects of the GAP processor regarding the power consumption are discussed, where these aspects have to be translated into a power

simulation in the future:

- As mentioned above, the simple frontend of the GAP processor saves a lot of power dissipated in the out-of-order instruction window as well as reorder buffer by searching for instructions with no data dependency and writing the results back correctly.

- During the loop state there is no need to fetch the instructions again from the cache. The array stays active, whereas other stages stall saving a lot of dissipated power in the frontend. For multimedia applications, it is a tremendous advantage, since more than the half of the execution time is usually spent in executing loops. Moreover, only active functional units consume power whereas others are gated to keep the power consumption per instruction at its minimum.

- The configuration layers are also introducing a disciplinary method to get the benefit of code spatial locality for both loop and functions. This increases again the time of execution inside the array—more than 80% of the overall execution time—and saves more energy by relieving the frontend.

- The register file access is an important factor in power consumption [88]. GAP architecture cope with this challenge by reducing the accesses to the register file. Instead of accessing the register file at almost every clock cycle as in pipelined processors, the GAP accesses the register file only twice during a configuration phase (when the execution in the array starts and finishes). In loop state is even no need to access the register file since the intermediate results can be bypassed to temporary registers at the top of the columns.

- Another factor of power consumption is that no clock tree must be applied to the FUs during the loop state, since the execution is asynchronous.

6.2.3 Software Optimization

The GAP processor executes programs that are generated by common compilers like GCC. Thus, additional software modification for the generated code is not necessary. However, some software optimization techniques can be applied to enhance the performance and increase the utilization of the FUs.

R. Jahr has implemented several software optimizations to the GAP like generating the code with predicated execution technique. By converting branches to predicates, the software creates larger regions without control flow changes inside it. This leads to a big and non-interrupted configuration phases in the grid. A conversion of the *if, then, else* control switch to a kind of predicted block is also presented in [89]. This is another technique that used to speculatively execute the block of code after an *else* control instruction by shifting this block to the preceding block with an additional appropriate predicate. Doing this places the instructions in the unallocated FUs and pre-executes them earlier. Thus, when the execution evaluates the branch instruction, the results of the executed block can be either considered or discarded based on the branch evaluation.

Software optimization of the executed code for coarse grained reconfigurable processors contributes to a better performance. Many software optimizations can be applied to the loops to increase the ILP and to gain more utilization of the processing elements. Many compiler optimizations can be implemented like, loop unrolling, pipelining, peeling/flattening, and function-inlining, which increase the parallelism and enhance the performance.

Loop unrolling and pipelining must take into account the dimensions of the array in order to achieve an optimal placement. On the other hand, it must take the number of iterations into account to avoid replicating non-executable code.

A more promising software optimization can also be done to manage the configuration layers. The current replacement strategy of the layers is implemented in the

hardware and it follows a simple FIFO strategy, which results in a non-optimal management. A more sophisticated method can be implemented in the software to manage the replacement based on the factors. The cost function should take into account the number of instructions in each layer, the number of expected executions in the future of each layer. This requires also a hardware interaction to evacuate the specified layer if needed.

List of Tables

List of Figures

Bibliography

[1] R. Murphy, "On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance," *IEEE International Symposium on Workload Characterization*, 2007.

[2] R. Hartenstein., "Coarse Grain Reconfigurable Architectures," *Asia and South Pacific Design Automation Conference*, 2001.

[3] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. H. Burrill, R. G. McDonald, and W. Yode, "Scaling to the End of Silicon with EDGE Architectures," *IEEE Computer Journal*, vol. 37, no. 7, pp. 44–55, 2004.

[4] M. B. Taylor, J. S. Kim, J. P. Amarasinghe, and A. Agarwal, "The Raw microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro journal*, vol. 22, no. 2, pp. 25–35, 2002.

[5] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. Filho, and V. C. Alves, "Design and Implementation of the MorphoSys Reconfigurable Computing Processor," *Journal of VLSI Signal Processing Systems*, vol. 24, no. 2–3, March 2000.

[6] B. Mei, S. Vernalde, D. Verkest, H. D. Man, , and R. Lauwereins., "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," *Field-programmable Logicand Applications Conference*, 2003.

[7] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[8] E. M. PANAINTE, K. BERTELS, and S. VASSILIADIS, "The Molen Compiler for Reconfigurable Processors," *ACM Transactions on Embedded Computing Systems*, vol. Vol. 6 and No. 1 and, February 2007.

[9] W. M. Johnson, W. M. Johnson, and W. M. Johnson, *Super-Scalar Processor Design*. book, 1989.

[10] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single Instruction Stream Parallelism is Greater Than Two," *SIGARCH Computer Architecture News*, vol. 19, no. 3, pp. 276–286, 1991.

[11] "PowerPC 750 RISC Microprocessor Technical Summary." [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_750FX_Microprocessor

[12] D. B. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro Journal*, vol. 16, no. 2, pp. 8–15, 1996.

[13] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Developement*, vol. 11, no. 1, pp. 25–33, 1967.

[14] R. Collins and G. Steven, "Instruction Scheduling for a Superscalar Architecture," *EUROMICRO Conference*, vol. 0, p. 0643, 1996.

[15] B. Appelbe, R. Das, and R. Harmon, "Instructions Scheduling for Highly Super-

scalar Processors," Tech. Rep., 1997.

[16] S. R. Kunkel and J. E. Smith, "Optimal Pipelining in Supercomputers," *SIGARCH Computer Archiecture News*, vol. 14, no. 2, pp. 404–411, 1986.

[17] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *ASPLOS-III: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, 1989.

[18] M. B. Josephs and S. M. Nowick, "Scanning the Technology: Applications of Asynchronous Circuits," *Proceedings of the IEEE*, pp. 223–233, 1999.

[19] M. Krstić, E. Grass, F. K. Gürkaynak, and P. Vivet, "Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook," *IEEE Design and Test*, vol. 24, no. 5, pp. 430–441, 2007.

[20] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical Design of Globally-Asynchronous Locally-Synchronous Systems," *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, p. 52, 2000.

[21] P. Wielage, "Clock Synchronization through Handshake Signalling," *ASYNC '02: Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems*, p. 59, 2002.

[22] A. Iyer and D. Marculescu, "Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors," *ISCA '02: Proceedings of the 29th Annual International Symposiumon Computer Architecture*, pp. 158–168, 2002.

[23] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains

with Dynamic Voltage and Frequency Scaling," *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, p. 29, 2002.

[24] Y. Zhu, D. H. Albonesi, and A. Buyuktosunoglu, "A High Performance, Energy Efficient, GALS Processor Microarchitecture with Reduced Implementation Complexity," *International Symposium on Performance Analysis of Systems and Software*, pp. 42–53, 2005.

[25] Y. Li, Z. Wang, J. Ruan, and K. Dai, "A Low-Power Globally Synchronous Locally Asynchronous FFT Processor," *HPPC: Workshop on Highly Parallel Processing on Chip*, pp. 168–179, 2007.

[26] A. M. Scott, M. E. Schuelein, M. Roncken, J.-J. Hwan, J. Bainbridge, J. R. Mawer, D. L. Jackson, and A. Bardsley, "Asynchronous on-Chip Communication: Explorations on the Intel PXA27x Processor Peripheral Bus," *ASYNC-2007: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 60–72, 2007.

[27] R. T. G. Estrin. B. Bussell and J. Bibb, "Parallel Processing in a Restructurable Computer System," *IEEE Transactions on Electronic Computers*, vol. 6, pp. 747 – 755, Dec. 1963.

[28] R. Hartenstein, "The Microprocessor Is No Longer General Purpose: Why Future Reconfigurable Platforms Will Win," *Proceedings of the IEEE Second Annual International Conference on Innovative Systems in Silicon*, pp. 2 – 12, 1997.

[29] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor With A Reconfigurable Coprocessor," *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, p. 12, 1997.

[30] R. B. Jr, P. M. Athanas, and M. D. Musgrove, "Colt: An Experiment in Wormhole

Run-Time Reconfiguration," *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic. SPIE*, pp. 187–194, 1996.

[31] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[32] H. J. Kim and W. H. Mangione-Smith, "Factoring Large Numbers with Programmable Hardware," *FPGA '00: Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, pp. 41–48, 2000.

[33] K. H. Leung, K. W. Ma, W. K. Wong, and P. H. W. Leong, "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor," *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 68, 2000.

[34] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Generation of Design Suggestions for Coarse-Grain Reconfigurable Architectures," *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 00th International Workshop on Field-Programmable Logic and Applications*, pp. 389–399, 2000.

[35] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," *Proceedings of the Conference on Design, Automation and Test in Europe*, 2001.

[36] *www.xilinx.com*.

[37] E. Ramo, J. Resano, D. Mozos, and F. Catthoor, "Memory Hierarchy for High-Performance and Energy Aware Reconfigurable Systems," *Computers and Digital Techniques, IET*, 2007.

[38] S. A. guccione, "Multicore Devices: A New Generation of Reconfigurable Architectures," *Engineering of Reconfigurable Systems and Architectures*, 2008.

[39] D. Cherepacha and D. Lewis, "A Datapath Oriented Architecture for FPGAs," *Proceedings of FPGA94: Field Programmable Gate Arrays conference, Monterey, CA, USA*, February, 1994.

[40] R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659–681, 2006.

[41] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Applications," *FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pp. 135–143, 1999.

[42] H. Singh, M. hau Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Transactions on Computers*, vol. 49, pp. 465–481, 2000.

[43] D. C. Chen, "Programmable Arithmetic Devices for High Speed Digital Signal Processing," 1992. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/2033.html

[44] A. Yeung and J. Rabaey, "A Reconfigurable Data-driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms," *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, vol. vol.1, pp. 169 – 178, 1993.

[45] H. Diab and I. Damaj, "Cyclic Coding Algorithms Under MorphoSys Reconfigurable Computing System," *in Advanced Engineering Software*, vol. 34, no. 2, pp. 61–72, 2003.

[46] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler, "A Design Space Evaluation of Grid Processor Architectures," *MICRO 34: Proceedings of the 34th*

Annual ACM/IEEE International Symposium on Microarchitecture, pp. 40–51, 2001.

[47] PACT XPP Technologies, July 2006, $http://www.pactxpp.com/main/download/XPP-III\_overview\_WP.pdf$.

[48] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal, "Baring it all to Software: The Raw Machine," Cambridge, MA, USA, Tech. Rep., 1997.

[49] R. W. Hartenstein and R. Kress, "A Datapath Synthesis System for the Reconfigurable Datapath Architecture," in *Proceedings of the 1995 Asia and South Pacific Design Automation Conference, ASP-DAC*, 1995.

[50] E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157–166, 1996.

[51] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable Multimedia Array Coprocessor," *IEICE Transactions on Information and Systems*, pp. 389–397, 1998.

[52] A. Alsolaim, J. Starzyk, J. Becker, and M. Glesner, "Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems," *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 205, 2000.

[53] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD- Reconfigurable Pipelined Datapath," *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 126–135, 1996.

[54] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor,

and R. Laufer, "PipeRench: a Coprocessor for Streaming Multimedia Acceleration," *Proceedings of the 26th International Symposium on Computer Architecture*, vol. 27, no. 2, pp. 28–39, 1999.

[55] J. Rabaey, "Reconfigurable Processing: the Solution to Low-power Programmable DSP," *on ICASSP-97 the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. vol. 1, pp. 275 – 278, 1997.

[56] D. Burger and T. Austin, "The SimpleScalar Tool Set and Version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, June 1997.

[57] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free and Commercially Representative Embedded Benchmark Suite," *4th IEEE International Workshop on Workload Characteristics*, pp. 3–14, 2001.

[58] D. I. Lehn, K. Puttegowda, J. H. Park, P. Athanas, and M. Jones, "Evaluation of Rapid Context Switching on a CSRC Device," *Proceedings of the second conference On Engineering Of Rconfigurable System And Algorithms (ERSA). CSREA*, pp. 209–215, 2002.

[59] J. Noguera and R. M. Badia, "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling," *ACM Transactions on Embeded Computing Systems*, vol. 3, no. 2, pp. 385–406, 2004.

[60] J. Resano, D. Mozos, D. Verkest, and F. Catthoor, "A Reconfiguration Manager for Dynamically Reconfigurable Hardware," *IEEE Design and Test*, vol. 22, no. 5, pp. 452–460, 2005.

[61] F. Bouwens, M. Berekovic, A. Kanstein, , and G. Gaydadjiev, "Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array," *ARC'07: Proceedings of the 3rd International Conference on Reconfigurable Computing*, 2007.

[62] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 24–35, 1996.

[63] "http://www.intel.com/products/desktop/processors/pentium.htm."

[64] F. Bouwens, M. Berekovic, B. D. Sutter, and G. Gaydadjiev., "Architecture Enhancements for the ADRES Coarse-Grained Reconfigurable Array," *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, 2008.

[65] S. W. Keckler, D. Burger, C. R. Moore, and R. Nagarajan., "A Wire-Delay Scalable Microprocessor Architecture for High Performance Systems," *IEEE International Solid-State Circuits Conference*, 2003.

[66] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP and TLP and and DLP with the Polymorphous TRIPS Architecture," *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[67] M. B. Taylor, W. Lee, J. E. Miller, and A. Agarwal, "Evaluation of the Raw microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," *Proceeding of 31st Annual International Symposium on Computer Architecture*, pp. 2–13, 2004.

[68] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs," *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pp. 795–805, 2002.

[69] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. Filho, and V. C.

Alves, "Design and Implementation of the MorphoSys Reconfigurable Computing Processor," *Journal of VLSI Signal Processing Systems*, 2000.

[70] G. S. Tyson, "The Effects of Predicated Execution on Branch Prediction," *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 196–206, 1994.

[71] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," pp. 45–54, 1992.

[72] A. Goldberg and J. Hennessy, "MTOOL: a Method for Detecting Memory Bottlenecks," *ACM SIGMETRICS Performance Evaluation Review*, vol. 19, no. 1, pp. 210–211, 1991.

[73] N. R. Mahapatra and B. Venkatrao, "The Processor-memory Bottleneck: Problems and Solutions," *Crossroads*, p. 2.

[74] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Effective Alternative to Large Instruction Windows," *IEEE Micro*, vol. 23, pp. 20–25, 2003.

[75] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[76] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, Feb. 2002.

[77] "Sun Technical Report for Niagara Machine." [Online]. Available: http://www.sun.com/processors/UltraSPARC-T2/

[78] S. Sethumadhavan, R. Mcdonald, R. Desikan, D. Burger, and S. W. Keckler, "Design and Implementation of the TRIPS Primary Memory System," *International Conference on Computer Design*, pp. 470–476, 2006.

[79] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Memory Bank Disambiguation using Modulo Unrolling for Raw Machines," in *Proceedings of the ACM/IEEE Fifth International Conference on High Performance Computing*, 1998.

[80] K. L. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[81] A. Aggarwal and M. Franklin, "Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors," in *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[82] V. V. Zyuban and P. M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Transactions on Computers*, vol. 50, pp. 268–285, 2001.

[83] R. Canal, J. manuel Parcerisa, and A. González, "Dynamic Code Partitioning for Clustered Architectures," in *International Journal of Parallel Programming*, 2001.

[84] R. Canal, J. M. Parcerisa, A. González, and J. Girona, "Dynamic Cluster Assignment Mechanisms," in *Proceedings of High Performance Computing Conference*, 2000.

[85] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically Managing the Communication-parallelism Trade-off in Future Clustered Processors," in *Proceedings of International Symposium on Computer Architecture*, 2003.

[86] F. Latorre, J. González, and A. González, "Back-end Assignment Schemes for

Clustered Multithreaded Processors," in *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, 2004.

[87] J. González, F. Latorre, and A. González, "Cache Organizations for Clustered Microarchitectures," *WMPI '04: Proceedings of the 3rd Workshop on Memory Performance Issues*, pp. 46–55, 2004.

[88] A. Lambrechts and P. Raghavan., "Energy-Aware Interconnect-Exploration of Coarse Grained Reconfigurable Processors," *Workshop on Application Specific Processors*, 2005.

[89] R. Jahr, B. Shehan, S. Uhrig, and T. Ungerer, "Static Speculation as Post-Link Optimization for the Grid Alu Processor," *in the 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)*, Italy, 2010.

[90] B. Shehan, S. Uhrig, R. Jahr, and T. Ungerer, "Reconfigurable Grid Alu Processor: Optimization and Design Space Exploration," *in the 13th EUROMICRO Conference on Digital System Design*, 2010.

Author Publications

[2] Basher Shehan, Sascha Uhrig,Ralf Jahr, and Theo Ungerer, ”Reconfigurable Grid Alu Processor: Optimization and Design Space Exploration”, *in the 13th EUROMICRO Conference on Digital System Design*, Lille, France, 2010.

[3] Sascha Uhrig, Basher Shehan, Ralf Jahr, and Theo Ungerer, ”The Two Dimensional Superscalar GAP Processor Architecture”, *in the International Journal on Advances in Intelligent Systems*, 2010.

[4] Ralf Jahr, Basher Shehan, Sascha Uhrig, and Theo Ungerer, ”Static Speculation as Post-Link Optimization for the Grid Alu Processor”, *in the 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)*, Italy, 2010.

[5] Basher Shehan, Sascha Uhrig, Ralf Jahr, and Theo Ungerer, ”Optimization and Evaluation of the Reconfigurable Grid Alu Processor”, *in the 15th CSI Symposium on Computer Architecture and Digital Systems (CADS)*, Tehran, 23-24 September 2010.

[6] Basher Shehan, Sascha Uhrig, Ralf Jahr, and Theo Ungerer, ”Enhancing the Grid Alu Processor for a Better Exploitation of the Functional Units”, *in the 17th*

International Conference Mixed Design of Integrated Circuits and Systems, Wroclaw, 24-26 June 2010.

[7] Sascha Uhrig, Basher Shehan, Ralf Jahr, and Theo Ungerer, "A Two-dimensional Superscalar Processor Architecture", *in the First International Conference on Future Computational Technologies and Applications (FUTURE COMPUTING)*, pp.608-611,Greece, 2009.

[8] Ralf Jahr, Basher Shehan, Sascha Uhrig, and Theo Ungerer, "The Grid ALU Processor", *in ACACES: Advanced Computer Architecture and Compilation for Embedded Systems*, L'Aquila Italy, 2008.

Basher Shehan

Personal Information:

Date of birth	14.05.1979
Place of birth	Hama, Syria
Nationality	Syrian

Education:

2007-2010	Academic Researcher in the Department of Computer Science at the University of Augsburg, Germany.
2005-2007	Master study at Kaiserslautern Technical University, Germany Master Thesis: ”Implementing a Formally Verified IEEE-754 Floating Point Unit with LISA”.
1998-2003	Diploma study in the Computer Engineering Department at the EIT Faculty, Aleppo - Syria. Diploma Thesis: ”Hiding of Information in Digital images for the Purpose of Network Security”.
1994-1998	High school: Hama, Syria
1989-1994	Primary School: Hama, Syria